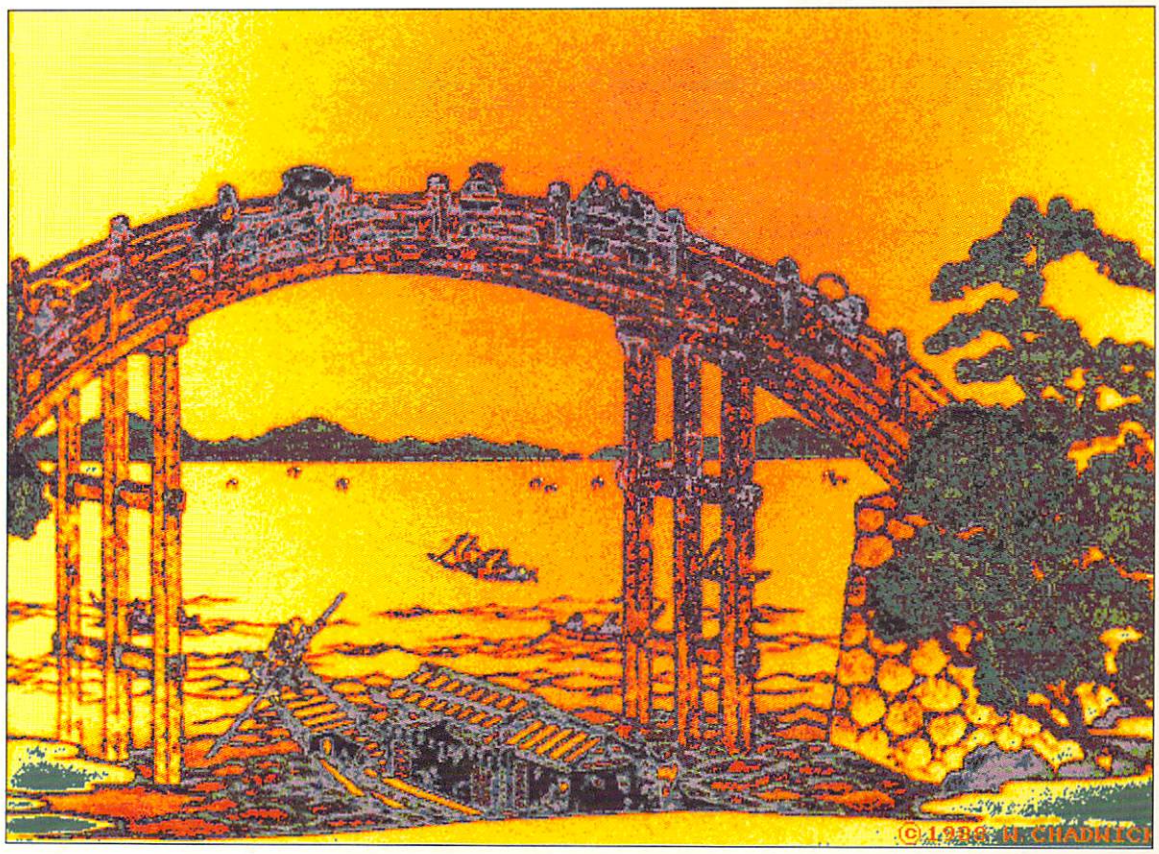
JULY 1989: VOLUME 2, ISSUE 4

THE MAGAZINE FOR AMIGA PROGRAMMERS

Canada $4.25
USA $3.50

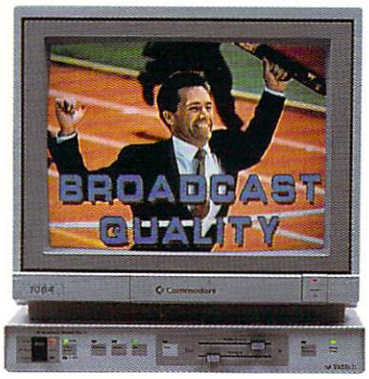
FOR THE Amiga Transactor

- Interrupts: the untold story *by Carl Sassenrath*
- Getting a handle on file systems, part 3 *by John Toebes*
- Mastering The Raster: An Amiga video production *by Derek Grime*
- Computer publications: pompous, verbose and illiterate *by Dick Barnes*
- Assembly language programming *by Jim Butterfield*
- Writing a compiler - part 1 in a series *by Chris Gray*
- Good documentation: some standards *by Rick Stevens*
- *WarpText* in Modula-2 *by Anthony Bryant*
- Cubic spline routines *by Eric Giguère*
- Hard drive usage tricks *by Jay Gross*
- **Reviews:** LightBox, GFA BASIC 3.0, Turbo Silver user manual



Heading Home by William Chadwick

0 29074 12122 07



Coming To A Screen Near You.

If you're ready to become your own studio producer, there's only one desktop video system that will bring NTSC and S-VHS to your Amiga's screen. The Scanlock system by VidTech. Other systems give you a genlock that's "either-or." They either support NTSC or S-VHS. But not both.

Two For One

With Scanlock, you get two genlocks for the price of one. So you can migrate from NTSC to the newer S-VHS format without investing in another system. We also offer a Scanlock model that is compatible with the International PAL TV format, including its new S-VHS version.

Scanlock's dual-genlock system gives you broadcast-quality video at consumer prices. And our system offers all the bells and whistles, including front-panel fingertip controls, multiple monitor connections for viewing and editing, and independent fade controls for both reference and Amiga videos. If your Amiga is supporting a load of peripherals, we provide a connection that lets you use an optional external power supply.

Scanlock is a breeze to use. A "glitch-free" vertical interval switch lets you insert, reverse and remove reference video and Amiga graphics at the touch of a finger. And you can switch between NTSC, S-VHS and analog RGB formats without moving cable connectors. When you're done editing, you can resume normal Amiga operation with the flip of a switch.

Don't Get Locked Out

So if you're ready to become your own producer, choose the desktop video system that doesn't lock you into one genlock — the Scanlock system by VidTech. Coming to a screen near you.

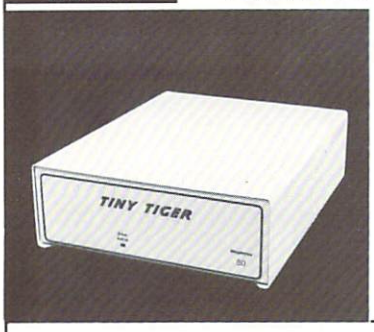
Call or write us today: VidTech International, Inc., 2822 NW 79th Ave., Miami, Florida 33122. Telephone 800-727-2261 or 305-477-2228. Fax 305-591-1651. Prices:NTSC-$995, PAL-$1,095. Distributed by Southern Technologies, Dallas, Texas.

VidTech

M.A.S.T.
Technical Excellence

"The Matchbox Collection" A POWERHOUSE OF PERIPHERALS.

MASSIVE MASS STORAGE FOR THE AMIGA 500/1000/2000



Maybe you want the unrivalled FLEXIBILITY of TINY TIGER — our UNIQUE standalone SCSI system, compatible with all Amigas.

POWERUSERS may prefer FIREBALL for their Amiga 2000. This true DMA, DRIVE ON A CARD, is designed to burn the competition on **price** and **performance**.

All MAST hard drive systems utilize State Of The Art, 3.5" SCSI drive technology. Both TINY TIGER and FIREBALL include a BUDGET and PERFORMANCE PLUS line of drives.

The BUDGET range comprise Seagate 30 & 50 megabyte drives.

The PERFORMANCE PLUS range feature FUJITSU 3.5" SCSI mechanisms — an indication of the reliability you should expect from this product. Naturally as another quality MAST product, it is SMALL, LOW POWER and covered by our ONE YEAR limited warranty.

PERFORMANCE PLUS drives are available in 45, 90, 135, and 180 megabyte configurations. They are voice coil, fast, and feature their own 24k byte RAM buffer for even faster access. The 45 meg mechanism is SUPERSMALL — at only 1" high it is the same size as our miniscule UNIDRIVE floppy.

TINY TIGER™
THE PURR-FECT SCSI SYSTEM

IF YOU CAN PLUG IN A PRINTER
THEN YOU ALREADY KNOW HOW TO
INSTALL A TINY TIGER
— Simple as that!

★ includes SCSI interface, drive and power supply
—all in one neat little case

★ drive is preformatted with workbench 1.3 installed
—power on and away you go

★ plugs into A500/A1000 and A2000
—great if you upgrade and ideal for moving software from one computer to another

★ connects thru 8 bit printer port
—a natural for SCSI — its 8 bit as well

★ fast — sure is — try this:
copy all of workbench 1.3 from RAM: to RAM: — you can't get much faster. It should take 22.5 secs. To do the same from Tiny Tiger to RAM:=27.6 secs. IN REAL WORLD TESTS — TINY TIGER PERFORMS.
— at lowest price per megabyte

★ printer is passed thru — transparently

★ pass thru for additional SCSI devices

★ TINY TIGER is attached by flexible cable and can be located in a convenient position
— DOES NOT OBSTRUCT USE OF A500 KEYBOARD

★ Bus expansion is left free for other important peripherals
— like memory

FIREBALL™
HARD DRIVE ON A CARD FOR THE AMIGA 2000

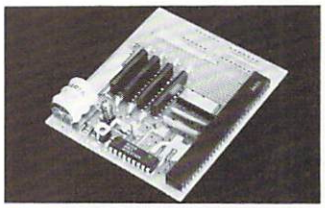
★ SCSI INTERFACE AND DRIVE
★ TRUE DMA
★ AUTOROOT **30 meg to 180 meg**
★ VERY FAST
★ LOW COST
★ INDUSTRY STANDRAD 50 meg CONNECTOR ON SCSI PASSTHRU

CAPACITY	FIREBALL	T/TIGER
30 meg	$ 599	$ 629
50 meg	$ 669	$ 699
45 meg	$ 799	$ 799
90 meg	$1099	$1129
135 meg	$1369	$1399
180 meg	$1569	$1599

REMEMBER — T/T PLUGS INTO ALL AMIGAS FOR A CLEAN, UPGRADE PATH — DON'T LIMIT YOURSELF

★ BUY TINY TIGER ★

MICROMEGS™ 512 K RAM AND BATTERY BACKED CLOCK



A500 ONLY $169

This alternative to the A501 is only half the size and uses a fraction of the power — important for power conscious A500 owners.

THE ULTIMATE EXTERNAL FLOPPY DRIVES

UNIDRIVE

★ SUPER SLIM LINE
★ VERY LOW POWER
★ NO CLICKING
★ MAY BE SWITCH DISABLED
(TWIN DRIVE HAS TWO SWITCHES)
★ QUALITY FUJITSU MECHANISM
★ POWERED FROM AMIGA™ OR OPTIONALLY AVAILABLE PLUGPAK

UNIDRIVE $159

TWINDRIVE

★ 1 YEAR LIMITED WARRANTY
★ UPGRADE POLICY—BUY A UNIDRIVE AND HAVE IT FACTORY UPGRADED TO A TWINDRIVE ANYTIME
★ REMOVABLE CABLE
★ ROBUST BEIGE METAL CASING
★ HINGED DUST COVER
★ QUIET OPERATION

TWINDRIVE $299

MEMORY TO GO!
NEED MEMORY FOR YOUR A500/A1000
TRY THIS — MINIMEGS™

2 MEGABYTES OF EXTERNAL RAM in a very small package

NEW LOW PRICES

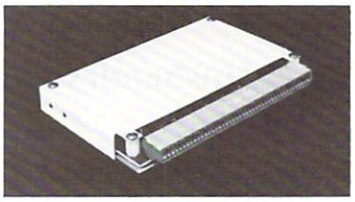


PHOTO OF A500 VERSION

- Auto Configure
- Low Power
- 1 Meg DRAM's
- RAM Tachometer
- Auto Configure LED
- Very Small Size
- Zero Wait States
- Compatible with A501

512K $329 **1 MEG $489**
2 MEG $779 (add $10 for all A1000 versions)

M.A.S.Terprint™ $1695

300 D.P.I.
DESKTOP PAGE PRINTER

- FUJITSU ENGINE
- LED TECHNOLOGY
  — for increased
  — reliability
- SMALL FOOTPRINT
  —only 16½"x16½"
- DUAL PAPER FEEDER
- HP-PLUS®

M.A.S.Terprint PS™ $3495

This is the POSTSCRIPT® version of the page printer. It has 35 fonts and 2 meg of RAM (optional 4 meg). Additional POSTSCRIPT font cartridges can be plugged into the printer.

IMPORTANT: MASTerprint can be upgraded to MASTerprint PS anytime — lets you save up for POSTSCRIPT.

AMIGA 2000 OWNERS
OUR NO-CLICK INTERNAL DRIVE IS A GREAT DEAL AT $109

MAST PERFORMANCE PLUS HARD DRIVES AND ALL OUR FLOPPY DRIVES FEATURE FUJITSU MECHANISMS

FROM JAPAN'S LEADING COMPUTER COMPANY

MEMORY & STORAGE TECHNOLOGY INC.

M.A.S.T. U.S.A.
3881-E Benatar Way
Chico, CA 95928
(916) 342-6278 Tel.
(916) 342-6281 FAX

M.A.S.T. U.K.
Unit 1, Dippen
Brodick, Arran
Scotland, KA278RN
(077082) 234

M.A.S.T. AUSTRALIA
178 Pacific Hwy.
St Leonards NSW 2065
(02) 436-2600 Tel.
(02) 437-4919 FAX

Amiga, A500, A501, A1000, A2000, Fujitsu HP+ and Postscript are trademarks of their respective companies. All prices and specifications subject to change without notice.

Volume 2, Issue 4

Publisher
Antony Jacobson

Vice-President Operations
Jeannie Lawrence

Assistant Advertising Manager
Mike Grantham

Editors
Chris Zamara
Nick Sullivan
Tim Grantham

Assistant Editor
Malcolm O'Brien

Contributing Writers
Steve Ahlstrom
Richie Bielak
Ben Blish
Daniel Brookshier
Anthony Bryant
Jim Butterfield
Betty Clay
Don Curtis
Matt Dillon
Andy Finkel
Brad Fowles
Eric Giguère
Cathryn Graham
Eric Graham
Chris Gray
Derek Grime
Jay Gross
Mike Hubbartt
James C. Lawrie
Rob Peck
Larry Phillips
Dan Schein
Steve Simpson
Richard Stevens
John Toebes
Victor A. Wagner
Doug Walker
Clark Williams
James D. Williams
Doug Winger
David Wood

Cover Artist
William Chadwick

for the AMIGA Transactor
The Magazine for Amiga Programmers

Interrupts: The Heart Of Exec — 19
by Carl Sassenrath
Carl Sassenrath *wrote* the interrupt code in the Amiga; in this article, he tells you how to use it.

A Vile Of Illiteracies — 24
by Dick Barnes
Our curmudgeon attacks computer literature - and provides some hilarious examples.

Getting A Handle On File Systems, Part 3 — 29
by John Toebes
An entire file system, compatible with AmigaDOS, is presented to illustrate handler concepts.

Effective Documentation — 36
by Rick Stevens
Standards for manual writers.

Cubic Spline Interpolation — 42
by Eric Giguère
Cubic splines let you represent complex curves with just a few points. Eric provides routines in C.

Debugging With Metascope — 48
by James C. Lawrie
A tutorial shows techniques for getting right to the problems in your code, using MetaScope.

Hard Disk Husbandry — 54
by Jay Gross
Now that you have a hard drive, what do you do with it? Here are some tricks, tips and techniques.

Programming In M2AMIGA — 57
by Anthony Bryant
Building on last month's article on interfacing with assembly, Anthony hooks up *WarpText*.

Toy - A Small Compiler for the Amiga — 62
by Chris Gray
Part 1 of a series in which Chris Gray develops a compiler before your very eyes!

Departments and Columns

Amiga Dispatches 5

by Don Curtis

The great debate: how to implement the HELP key? Also, the future of Amiga WordPerfect.

Mastering The Raster 8

by Derek Grime

A commercial video production, from start to finish: what it takes to complete a job

Assembly Language Programming 12

by Jim Butterfield

Jim delves further into the DOS routines and a variety of new topics

Hard Copy 71

by Tim Grantham

A look at the manual that comes with *Turbo Silver* teaches some lessons on what *not* to do.

Reviews

LightBox from R & DL Productions 74

GFA BASIC 3.0 from Antic Software 76

About the cover: *Heading Home* by William Chadwick

William Chadwick does freelance artwork on the Amiga apart from his full-time job restoring Tiffany lamps and windows. He became interested in Japanese art while in Japan in 1969 doing stage lighting on tour with the band *The Monkees*. The cover picture originated from several ancient Japanese prints, which were digitized with *Digi-View*. Some image processing was done on the pictures with *PIXMATE*, and the resulting images were brought into *Deluxe Paint II*, where pieces of them were turned into 'brushes' and joined together to form the final picture. DPaint was also used to brilliantly colour the grey tones in the black and white image.

Transactor for the Amiga is published monthly by Croftward Publishing Inc., 10-85 West Wilmot Street, Richmond Hill, Ontario, L4B 1K7. ISSN# 0838-0163. Canadian Second Class Mail Registration No. 7690, Gateway-Mississauga, Ont. USPS Postmasters: send address changes to: Transactor, PO Box 338, Station C, Buffalo, NY, 14209.

Croftward Publishing Inc. is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names are registered trademarks of Commodore Inc.

Subscriptions: see order card at centre of magazine for complete ordering information and prices.

Send all subscriptions to: Transactor Subscriptions Dept. 10-85 West Wilmot Street, Richmond Hill, Ontario, Canada, L4B 1K7, (416) 764-5273. Note: Subscriptions are handled at this address ONLY. Subscriptions sent to our Buffalo address (above) will be forwarded to our Richmond Hill HQ. For best results, use the postage paid card at the centre of the magazine.

Quantity Orders: In Canada: Ingram Software Ltd., 141 Adesso Drive, Concord, Ontario, L4K 2W7, (416) 738-1700. In the USA: IPD (International Periodical Distributors), 11760-B Sorrento Valley Road, San Diego, California, 92121, (619) 481-5928; ask for Dave Buescher.

Editorial contributions: writer's guides are available by writing to the Richmond Hill address above. Write to the editorial department (same address) with article proposals or articles for consideration.

All material accepted becomes the property of Croftward Publishing Inc., except by special arrangement. All material is copyright by Croftward Publishing Inc. Reproduction in any form without permission is in violation of applicable laws. Write to the Richmond Hill address for a writer's guide.

The opinions expressed in contributed articles are not necessarily those of Croftward Publishing Inc. Although accuracy is a major objective, Croftward Publishing Inc. cannot assume liability for errors in articles or programs. Programs listed in Transactor for the Amiga, and/or appearing on Transactor for the Amiga disks, are copyright by Croftward Publishing Inc. and may not be duplicated or distributed without permission.

Transactor for the Amiga is produced using Amigas and the following products:

Professional Page
Deluxe Paint II
Transcript
Digi-View

Final output by Vellum Print & Graphic Services, Inc., Toronto

Printed in Canada by Bowne of Toronto, Inc.

GFA-BASIC 3.0 for the Amiga

$139.95

GFA SYSTEMTECHNIK

Boldly goes where no BASIC has gone before.

- High-Speed Interpreter for easy program development
- Over 300 powerful commands
- FAST! – Execution times comparable to C
- In-line C and Assembler Commands
- Easy access to all Amiga libraries
- Extensive Amiga commands with submenus and built-in file requester
- Built-in Text Editor with syntax checking, procedure hiding and auto-indenting
- 400-page comprehensive manual
- Includes Run-Time Interpreter

Available NOW

from

Antic Software

544 Second Street, San Francisco, CA 94107

Call today to order:

800-234-7001

or see your local Amiga dealer.

(GF9200)

GFA-BASIC 3.0 is a trademark of GFA Systemtechnik, Germany; Antic Software and Antic are trademarks of Antic Publishing, Inc.; Amiga is a registered trademark of Commodore-Amiga Inc.

Amiga Dispatches

The HELP key and a challenge...

by Don Curtis
Copyright © 1989 Donald A. Curtis

Don Curtis is a police officer in Denver, Colorado, who has for the last three years been assigned to program design and development along with system design and maintenance of ten AT&T Unix 3B2 computers. In his spare time, Don is an assistant sysop on CompuServe's Amiga forums.

Larry Phillips (another of the assistant SYSOPs on CompuServe, and author of the column 'ViewPort' in this magazine) and I have some strong differences on how the HELP key should function. We argued about it in the Amiga Tech Forum on CompuServe, but decided to bring it into the pages of this magazine and let you, the readers, decide how things should work. Not that any of this will be 'official' or carry any weight with Commodore, but it should be thought provoking, and thus, useful.

A quick rundown of Larry's method of implementing the HELP function is that it should be context sensitive based on the mouse position and what the program was doing at the time. Essentially, the way I understand his methodology is that there is a universal 'help.handler'. Through this facility, when the user presses the HELP key, based on the user's preferences and the program's implementation, the system looks at where the mouse currently is, and gives HELP on that specific object whenever possible. The forms of help given can be program generated, via a 'help' text file the user creates, or a combination of both. One thing Larry definitely insists on is that when pressing the HELP key, the position of the mouse is paramount and *without* regard to whether the mouse is in the currently active window or not.

Generally, my idea is rather simple. First of all, the HELP key is just that, a key on the keyboard, and like all keystrokes (except mouse shortcuts) it should *only* apply to the currently active window. Second, by its very nature, help is specific to a particular program. The help information for Program A is of use only with Program A. Because of that, it's up to the program to offer help, not the system. Third, tying help to the mouse position can be a pain in the butt. I generally keep the mouse out of the way. If I hit the HELP key (like right now), I'd get help with the scroll bar, when all I wanted was the quick reference card to pop up. With Larry's system, at the least, I'd have to move the mouse out of the scroll bar into a 'generic' area to get my quick reference card. Finally, I don't like the idea of, for all intents and purposes, the manual being built into the program.

Most importantly, I feel the HELP key should always apply to the currently active window. Mouse actions always work on the active window, or make a window active: if I want to close a window, I move to the close gadget and click on it. All that does is make that window active, and informs the program the user has selected the close gadget. It's up to the program owning that window to actually do the close. Larry's idea is similar in this respect, except he doesn't want the keypress of the HELP key to make the window active, just to notify the program that the user pressed the HELP key.

How's this for an alternative approach? If help only goes to the active window and help is only under program control, not system control, then when I hit the HELP key, I get a requester asking what I want help with. There could be buttons for 'Generic Help', 'Help with Gadgets' or 'Help with Menus' (plus whatever else the author decided to implement). I select 'Help with Menus' and get a requester telling me to select the menu item where I need assistance. Since all gadgets, menu items, etc. are under program control, it's no problem for the program to offer help with 'Quit' rather than quit the program. I simply select the menu item just as I would when running the program. The program notes I'm asking for help, not asking for the selected action to take place, and gives me assistance. All with one hand too – no awkward holding of mouse and help key at the same time.

Of course, both implementations would require the program to offer that level of help. To be honest, I disagree with that much code and program space being used to repeat what's in the manual. I much prefer that program-generated help be more in the line of a quick reference card. I hit the HELP key, and a quick reference requester pops up giving the most commonly used keystrokes and listing the manual page for additional assistance. Piracy exists, people steal software. I see no reason to help them by providing an on-line manual with the program. I'm not saying program should penalize the honest user; a

quick reference card, built into the program with pointers to the manual page for more information should be sufficient. That's all most people want; they forget the keystroke(s) for a specific action, and just need a quick reminder.

Well, those are my ideas, and objections to Larry's, in a nutshell. It's up to you to decide how you want things implemented. I'd be interested in your thoughts on the subject. Larry and I are just two Amiga users; it's not important what we think as individuals. What we're really interested in is what the Amiga community thinks as a whole.

WordPerfect down the tubes? No!

On April 29th, WordPerfect Corp. announced in the Amiga Forum that all *new* development on the Amiga has been halted. The final release of v4.1 will be finished, bugs will continue to be fixed, and the list price will be dropped to about $250, but the rumored v6.0 full WYSIWYG (What You See Is What You Get) version has been scrubbed along with any other new version. They've pulled five of the Amiga programmers and placed them on other development projects. Apparently the rumoured idea that v6.0 would be developed on the Amiga, and then ported to the other machines was just that... a rumour.

There were several reasons given for this decision, the major reason being that the Amiga market hasn't paid its way. Not even close – in fact, they've lost the better part of a million dollars to this point. The money drain just simply had to be stopped. The loss wasn't always there: initially, the Amiga version of Word Perfect made money. In fact, the costs of development (about half a million dollars) were re-couped within a few months of its release. But after those first few months, the sales dropped dramatically, and continue to drop. The continuing costs are what ate up the profit. WordPerfect Corporation provides superior support of their products over toll-free phone numbers, which is a high ongoing cost. They also had a large development team working on new products like Library and PlanPerfect (which has now been dropped). The market simply didn't support that level of expenditure.

Seventy percent of the Amigas in the world are outside the US and Canada, so, in an effort to stem the losses, WordPerfect tried selling a German version. Sales there were horrible. They didn't even recover the cost of translation. While there are other areas of the world where the Amiga is sold, failing so miserably in Germany left the prospects for expansion into those markets a poor bet. So while the Amiga's sales are very close to one million units (in fact, one million units have shipped from Commodore), the potential market for WordPerfect was limited to about 300,000 users.

The question that remained was, what caused the bottom-line drop? WordPerfect Corp believed that perhaps the Amiga market wasn't interested in a text oriented product, that what they needed was a graphics oriented program, one that would show graphics and fonts on-screen. They also admitted that they made mistakes. They failed to advertise sufficiently, and what advertisements there were didn't stress the features of the package, only that it was WordPerfect. They also admit that perhaps they brought it to the market before it was really ready. More testing should have been done so it wasn't so buggy when it hit the shelves.

On the other hand, the list price may have been too high for the Amiga market. Their statistics show that while the Amiga itself continues to sell well, the majority of sales are of the A500, and WordPerfect just doesn't run well on an unexpanded A500. Not only that, the list price of WordPerfect doesn't lend itself to the price ranges A500 owners tend to look into.

Commodore has been mentioned for lack of support given to WordPerfect Corp. I have no knowledge of what support was asked and what was received... I'm just passing it on as another one of the issues discussed.

And of course, the final issue was that no piece of software will capture the entire market. WordPerfect is a very powerful professional-level word processor. While it's more than adequate for ripping off a quick letter to Mom and Dad, its real use comes in the professional writer arena. Someone who writes for a living would need its features, and those who don't can (and do) get along with lesser (and cheaper) products. That limited the market potential even more.

A brand-name product like WordPerfect is needed in the Amiga market. It's needed because it *is* WordPerfect, not some clone. Like it or not, it gives the Amiga an air of legitimacy. I was at a meeting the other day, discussing software used at work. A question was raised about why the department used so many different types of databases. The reasons were many, but broke down to cost and unneeded power. Well, after lots of discussion, it was decided that we would no longer buy anything except dBase. Not because everyone needed it, but because it was standard, in fact, it was 'the' standard. Once someone learned how to use it in one office, they could go to another office and use it. They wouldn't have to learn a whole new software package. We could send everyone to a single school to learn how to use dBase, and those schools were readily available.

That's why we need WordPerfect. It is 'the' standard word processor in many, many offices. If you learn WordPerfect at work, you can use it at home with little, if any, change. For some people, WordPerfect was the last reason they had to hold onto their MS-DOS machines. When it became available on the Amiga, the MS-DOS machine went into the closet for good.

Well, the reaction to that announcement was long and loud. As I said, WordPerfect Corp has excellent support. That included Pete Peterson, a member of the board of directors, not only being the one to post the message, but sticking around for weeks afterward and listening to the members of the Forum make suggestions, sympathize, vent their frustrations or just plain blast him. He stuck in there, answered messages, took the time to explain their decision and admit that mistakes were made.

Not only that, but he really did listen to the membership. He listened so well, in fact, that he changed his mind. He became convinced that the Amiga market wasn't quite as he saw it. The majority of users who took the time to leave a message said that they wanted a solid, bug-free product. It didn't have to have every possible bell and whistle. It didn't have to be full WYSIWYG – a preview mode would be fine to verify what the final output would look like. It didn't even need graphics; it wasn't intended as a desk top publishing program, it only needed to be a damn good word processor.

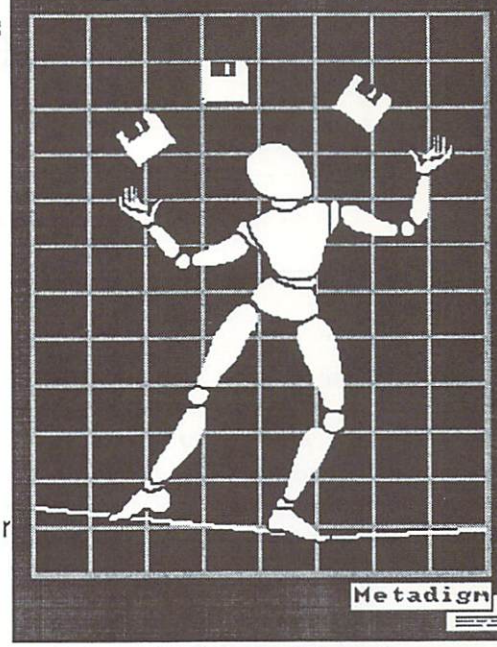
At a meeting with the Amiga programmers on May 8th, it was decided to continue with Amiga development. This development would be at a lower level than in the past: only two or three programmers would be working. They would work towards a v4.2 product with some enhancements. This would take approximately a year to complete. Some of the current plans for the future include, 4.2 file format, print preview, sorting, document comments, ARexx support, non-interactive spell checking and more. The future beyond that would depend on sales, but for the next 18 months or so, WordPerfect Corp will stick with the Amiga market.

That is perfectly reasonable. WordPerfect can't continue to lose money on the Amiga market at the rate they have in the past. They'll give the market a second chance, and we'll get a better product. Let's hope it works out well for both WordPerfect and the Amiga owner.

NEW Release ver 1.10

- Full Intuition based Interface
- Selective Restore
- Amiga wildcards
- Multiple indexing for greater **safety**
- Over 907K per floppy

saf-T-net



Hard Disk Backup

for your Amiga

available at your dealers **now**

or from (714) 955-2555

Metadigm, Inc. 19762 MacArthur Blvd. Suite 300; Irvine CA 92715

☆ ☆ ☆ ReSource ☆ ☆ ☆

Intelligent Interactive Disassembler for Amiga Programmers

"**ReSource** *makes full use of the Amiga windowing environment ... One of the features that distinguish* **ReSource** *is its ability to produce output which is compatible with most popular assemblers ... a bargain in this world of overpriced software." — Andre Marino*

ReSource is an interactive disassembler for the Amiga computer. Written entirely in assembly language, this is the most powerful disassembler available on *any* personal computer! Over 700 menu functions allow you to:

✔ Load from *any* file, read tracks, or disassemble memory directly.
✔ Save ALL, allowing you to continue disassembling later.
✔ Save to a ".asm" file, partial saves supported.
✔ Automated symbol creation:
JSR -$1E(A6) becomes JSR _LVOOpen(A6)
MOVE.L #$3EE,D0 becomes MOVE.L #MODE_NEWFILE,D0
Over 5000 symbols supported!
✔ Incredibly smooth scrolling.
✔ Full key rebinding supported. Load/save keytables.
✔ Up to 38 user-defined macros available. Load/save/overlay macros, looping/conditionals supported, variable execution speed.
✔ Numeric and text processing functions allow you to automate many things that used to make disassembling so slow!
✔ Search function uses full ARP wildcards. Search forwards, backwards, or use the special bi-directional search!
✔ The *fastest* text rendering available on the Amiga.
✔ Forward and backward referencing allows you to quickly examine referenced code/data, and return to your original position. Nested references supported.
✔ Automatically/manually create labels. Manually create/edit/delete labels, symbols, end-of-line comments, full-line comments.
✔ Special support for "C" programs, including "A4" register data referencing.

US$75 **If you're serious about disassembling code, look no further!** **A$90**

To order **ReSource V3.05**
Send check or money order
Credit cards accepted

The Puzzle Factory
P.O. Box 986
Veneta, OR 97487
U.S.A.
(503) 935-3709

Glen McDiarmid
28 Marginson St.
Ipswich, QLD
AUSTRALIA 4305
(07) 812-2963

Mastering The Raster

Putting the pieces together

by Derek Grime

Derek Grime is president of Beyond Graphics, a Toronto-based studio that uses the Amiga for a variety of graphics, animation and video applications. Derek can be contacted at Beyond Graphics, P.O. Box 1249, Station F, Toronto, Ontario Canada, M4Y 2V8.

Let's dream about the way things could be. A client walks into your studio with a finished storyboard. You glance through the scenes, noting that they require 2D and 3D animation, image processing, titling, colourization, digital video effects and a background music score. In this perfect world of the future, one software package handles the entire production. Within minutes of quoting on the job, you are actually working on it.

People who create videos with workstations and paint boxes are used to working in this fashion. For them it usually takes only one all-inclusive piece of software to produce an entire job. However, software of such scope costs many thousands of dollars. In fact, the price of one package is often more than a fully configured Amiga 2500.

Producing professional video and animation on the Amiga today is a little like trying to solve a puzzle with some of the pieces missing. Our style of desktop production requires that we stay on top of the software market – we have to know what each package offers and what each one lacks. We have to be well versed in both two- and three-dimensional animation. We have to have a feel for graphic design.

The best way to learn what to use is by watching others make mistakes; let me offer myself as an example. We'll examine a real job that my graphics company recently finished. In this case, the final video was quite successful; but the production itself was unusual because it was composed of a mix of unrelated styles. To stay on time and on budget required some crafty manoeuvering. Judge for yourself how it turned out.

The assignment

The client was with a small independent video production company. We had worked with them once before. They knew what services we offered so there was no need to show them our video portfolio. They had brought along a rough preliminary storyboard with scenes sketched out on it. We discussed the animation over a cup of coffee. The sequences were for an industrial video on worker safety. Some of the scenes were to be realistic, while others had to be more cartoon-like. The final result would be edited on three-quarter-inch tape. The deadline was tight but not unrealistic.

Basically there were four different scenes that required our services. The first needed two medical illustrations; the movement had to present the idea that the person was in pain. The second scene required an animated desk that would metamorphose into a different design. The third scene was of a UFO flying through space towards the Earth; the flying saucer had to stall and backfire before it finished its journey. This sequence also needed sound. The fourth and last scene required scrolling credits, like the introduction to the movie *Star Wars*.

You might be mulling this over, thinking about the ways that you would solve each of these problems. Keep in mind that the budget is always a big factor. Problems have to be solved quickly or you don't make any money. Sometimes the solutions you choose may not be the most elegant because of time constraints. Presenting the idea cleanly is all that matters.

Cutaway to the hand

Once we had a rough idea of what was needed, we had a second meeting. Since the original storyboard was very rough, we cleaned it up. A clean storyboard gives the customer a good representation of how the finished video should look. We also got more information on colour, timing, and stylistic considerations. Our quote was approved, and we started on the video.

The first sequence was the two medical illustrations. A cutaway drawing of a hand had to indicate areas where pain and stresses would develop after a long day's work. Artwork was supplied and, as it was from a government handbook, accuracy was important. The graphic would fade into view, hold for five seconds and then the animation would start and continue for another ten seconds.

Initially we considered drawing the hand in different positions. We decided twelve pictures would be needed to show a slight

movement, and to convey the idea of pain, which we originally envisaged as a series of radiating red lines.

With this in mind, we started by scanning in the artwork with Digi-view. High resolution was used to give the greatest amount of detailing. The supplied artwork was a photocopy, and there was a fair bit of surface garbage on the image. Surprisingly, the mess worked in our favor. In high-res the flaws from the original gave the computerized version a more realistic feel.

Our next step was to use *Pixmate* to prune the number of colours used. We always like to leave a few empty registers at the end of the palette for later use. Next we used the colour bias control to tint the image to a bloody red hue. The effect was gruesome. Once we were satisfied, we loaded the picture into *Deluxe Paint* for further cleanup.

The final still was very effective, and we worried that animating it would lessen the impact. After some discussion we tried a test animation using only colour cycling. Selecting a graduated series of bright reds, we overlaid them on top of the hand. When cycling was turned on we had our radiating pain animation. Choosing cycling over drawn animation saved us hours of studio time.

During the earlier stages of a production we keep the customer very involved. If they are available, we like to check all changes out with them until we get a good idea of what they need.

The client was happy with the results but requested minor changes. Not surprisingly, our picture was a little too grotesque; it *was* pretty disgusting. We lightened it up a little by brightening the colours in the palette. We added a background grid (the old staple) to give it more of a high-tech appearance. This made it look less like a still from a splatter movie.

Satisfied, we moved on to medical illustration number two. It was a full male figure with more of our patented cyclic pain animation. Looking through our anatomy text books gave us a range of repulsive source material, so we decided that a lighter approach was necessary.

We photocopied one of the figures to the correct size and then used our handy *Easyl* drawing tablet to trace around the picture. Freehand draw seemed too somber, so we opted for using the unfilled polygon mode. This gave our art a computeresque feel that the client seemed to want. Of course, we only traced half of the figure and then flipped it on the X axis to speed up the process. Illustration two was finished in record time and approved on the first showing.

To 3D or not to 3D?

The second sequence we had to complete was an animation of a transforming desk. Two pictures of desks were supplied and it was our task to change one into the other. The end result would show an uncomfortable work area becoming a pleasant place to work.

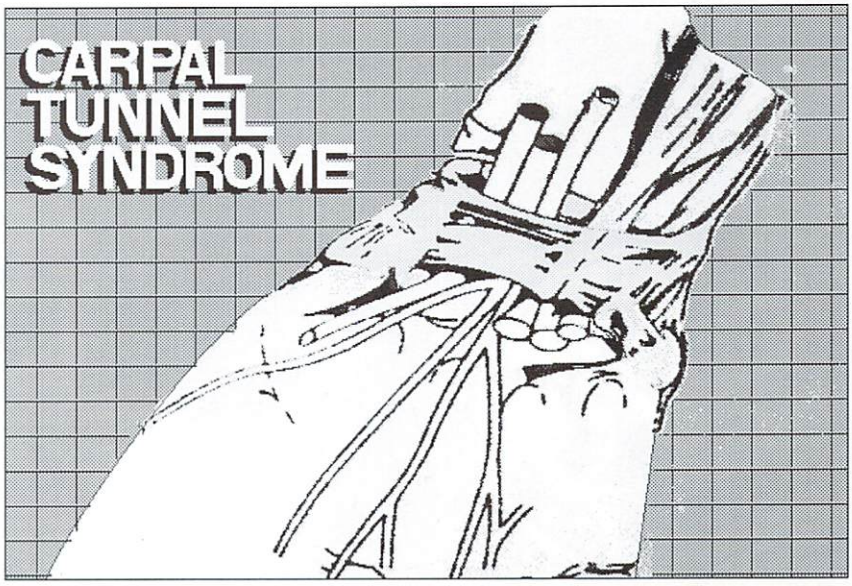
We initially considered using three-dimensional modeling software. The models of the desk could be created in *Sculpt/Animate 3D* and then hammered into the new shape. In theory the software would smoothly fill 'in-between' the two work areas. We could even add a rotation at the same time.

In practice, however, there were problems. Colour gave us difficulties: even a test in HAM mode made the desk seem dark and foreboding. Furthermore, the desks had to have detailing with office paraphernalia on top. An object as complicated as a typewriter takes a lot of computing horsepower to generate. We didn't have time to wait for the entire animation to render, so we decided to switch gears and cut our losses. Classical animation seemed the way to go.

Before starting to do the sequence over by hand, drawing each and every frame, we decided to get the client's opinion. What sort of animation did they want? After a short phone call it was agreed that the two pictures could be simply faded from one into the other. Why hadn't we thought of that? The solution was so simple that we had just dismissed it.

It's always wise to keep in constant contact with the customer in these matters. They are looking at your work from a different perspective and often make suggestions that you would never consider.

Scene three generated new problems. A flying saucer had to fly in from deep space and exit over the viewer's head. After a



The hand cross-section was digitzed from a photocopy and cleaned up

pause, it was to reenter from the right, stall its engine, backfire and fly down to the Earth. The background was a deep space scene that would remain static throughout. We were asked to make the scene playful.

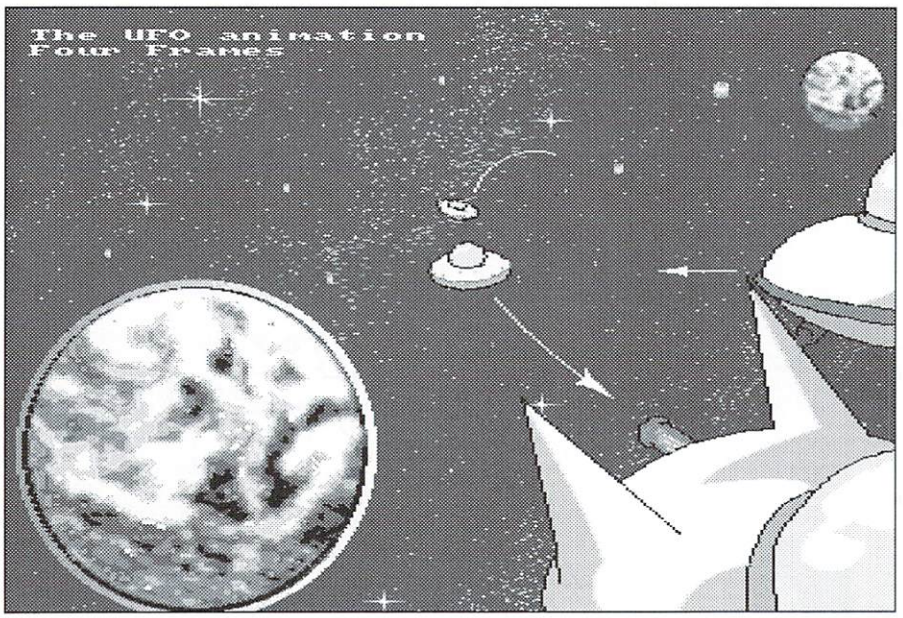
Again, we first looked at three dimensional animation. *Videoscape* seemed ideal. After designing a simplified ship we did some quick tests. The results were fine but they lacked life.

Three dimensional animation editors on the Amiga give the user minimal control over timing and arcing paths. I prefer to heavily weight any move at the beginning and end, making objects move in fits and starts. You can do this with any software by rendering far more frames than are actually necessary and then removing most of the middle body of the sequence. *Videoscape* will allow you to render only the parts of a move you require, a feature that really cuts down on system overhead.

The tests weren't quite what we had hoped for. Stylistically, our saucer looked too menacing. Rather than the playful appearance we had hoped for, our animation reminded us of pictures from the 'Mars Attacks' gum card series.
It was time for another call to the client. We don't ask for advice during these discussions, we just try to toss ideas around and hope that something clicks.

I was leaning towards drawing every frame by hand, and before I made that commitment I wanted to make sure I had a verbal okay. Classical animation sounded good to the customer, and we decided on a style over the phone.

I suggested a cross between nineteen-fifties Warner Brothers and 'The Jetsons'. Imagine trying to describe these tacky finned spaceships to somebody who had never seen an episode of the show before. Luckily, someone in the office knew what I was raving about so we got our approval.



Planning the flying saucer animation

Close up on classical animation

Once the background was drawn and the stage set, we had to decide on timing. For this we use a variation on the traditional animator's exposure sheet.

An exposure sheet is a chart that lists every second of a production, broken down into one-frame increments. There are thirty frames used per second in video production, but the Amiga can only flip drawings at that speed if they are in low resolution with only one bit plane (black and white). Our exposure sheet shows a breakdown to ten frames per second, as this gives us fluid movement without taxing the hardware.

Beside the frame count is a description of the action including all sound effects. These are written vertically on the page and fill the duration of time they will require in the finished piece. The individual drawings that correspond to the action are listed on the right hand side. Our exposure sheet showed that seventy-six drawings were needed. This was calculated after reducing an even higher total down by adding some repeated motions.

On paper we decided where the action would take place. Rough illustrations showed the main points of action in the scene and the paths that the objects would follow. Next, the key frames that show the extremes of every movement were drawn. The keys were then fleshed out with in-between drawings and then the in-betweens were in-betweened until the desired timing and smoothness was reached. It's a lengthy process. Until some new software arrives, all the in-betweens have to be done on paper using a light table to assist in the animating. (New software *has* arrived since: see the review of *Light Box* from R & DL Productions in this issue.)

After the drawings were completed, each one had to be traced into *Deluxe Paint* with our *Easyl* and then coloured. The coloured pictures were stored as individual IFF files and overlaid on the background one by one. The real advantage the computer brings is making the colouring of the individual cels much faster and allowing quick previews of the motion.

We had drawn the sequence in interlace resolution to give us the best trade between detail, speed and number of colours. We used only four bit planes, or sixteen colours, for every drawing to speed up the final animation.

When we were finished it was time to view the completed work. We first tried Mindware's original *PageFlipper* program. *PageFlipper* uses no compression and the results were too slow even when we ran them at full blast. To be fair, *PageFlipper* is really meant to be used in low resolution for real-time applications.

Our next experiment involved using the public domain program *MakeAnim* to compile a compressed 'anim' file. The results were still a little too slow and we lost all of our control over timing. With time running short, we turned to Mindware's newer flipper *Pageflipper +F/X*. PFX uses its own compression format, and whatever it is, it really works. Nothing else we used was nearly as fast.

We had also promised to add sound to this animation, or at least supply a cassette of timed sound effects. Using a sampler, we created a bunch of spacey sounds by 'poaching' some synth tracks and distorting them into something suitably cosmic. The backfire effect was actually a gunshot slowed way down, and the sound of an engine starting was modulated and overlaid with a heavy echo.

Sequencing the sounds was not as easy as creating them. We had hoped to use a sampler editor to compile a massive sample we could play as a loop. The editor we have doesn't allow the addition of dead air so we were unable to adjust the timing. Rethinking things, we decided to give trusty *Sonix* a try. By playing the samples as though they were instruments in a song, we got the results we needed. Adjusting the individual samples so that they stopped at the right point was tricky but not impossible. When the sound song was complete, the tempo could be adjusted up or down. While watching the animation on one machine, we tweaked the effects track on another Amiga close by and recorded it onto tape. The sound effects track matched the animation perfectly.

Crawling credits create chaos

The final sequence should have been the easiest to produce. Instead, it was the most difficult.

The script called for twenty-five lines of text to crawl up the screen. The point size would have to be twenty or greater to be legible from a distance. The opening crawl was to be reminiscent of *Star Wars*: a space background and lines of text crawling in perspective. When we took the assignment we explained to the client that achieving perspective type was difficult. We were told to do our best.

We prepared four IFF-format pictures in interlace resolution. Each one had several lines of text, centered on the screen. The idea was that there must be some way of animating the text up the screen in perspective. *Videotitler* was our first attempt. There is no automatic control of the perspective mode, so we eyeballed the settings as best we could. The test anims we prepared had the type dancing and wavering all over the screen. It reminded us of a roller coaster ride and was impossible to read without getting seasick. We considered *VideoEffects 3D* but abandoned it over the lack of fine timing control.

Next up was *DPaint II*, but the process of single stepping every frame and letting it render in perspective was too slow to be realistic. After several failures we discussed the possibility of the client using a digital effects unit in post production. We would supply the text scrolling smoothly up the screen, and the post house would move the type back into the distance. After the client phoned around to get some quotes he found the price of the extra unplanned post production to be over budget. Even a simple effect like this would cost hundreds of dollars.

We discussed our options. They agreed to accept a straight vertical scroll as long as we could place an animated perspective background of a star field behind the text. While this seems like a lot of work, it's actually quite an easy effect to get in DPaint using cycle draw.

We attacked the problem again. We could not find a single program that could scroll large type up the screen smoothly. We tried slide shows, but the images crawled too fast. We tried *The Director*, and while I'm sure it could have done the job, we didn't have the time left to experiment with it. We tried the titling program that comes with the A2300 Commodore genlock. It couldn't smoothly animate anything over ten point text. We were stuck.

Then I remembered a public domain program called *Startle.*, which scrolls text up the screen while showing an animated star field behind. It was exactly the effect we wanted but the scroll was a little too slow. Hobie Orris, the programmer of Startle, is a close friend. I called Hobie and asked if there was any easy way that Startle could be made faster. He suggested tusing four bitplanes for the background picture and scrolling the text in the fifth, and agreed to hack something together.

The deadline on the job was coming quickly. I phoned the customer and told him about the scheme. He was impressed that a custom program was being written especially for his production and gave his blessings. He was also finishing up the live action section of the video and that helped to buy extra time. Within a couple of days Hobie delivered the Son of Startle; the results were exactly what we had hoped for the first time we tried it. The job was finished.

Exit laughing

If we had created the animation and graphics for this job today we could have produced the entire job with *Deluxe Paint III* and a slideshow utility. There are still some large holes in the video productivity software on the Amiga, but they're being filled fast. One day we may even see the dream program for video pros, an all-inclusive piece of software that does everything you could ever wish for in two and three dimensions. As long as we're dreaming, it would be great if it could write the invoices and chase down overdue accounts as well.

One final note. We took the most pride in the title sequence: a seemingly simple credit scroll that took dozens of man hours to produce. When the client returned from the shoot, we previewed our masterwork for him. He loved it, but couldn't use it. In the two days he had been away, the script had been changed and the titles were no longer necessary. We would be paid, but the piece would never be seen. That's show biz...

Assembly Language, Part 5

A pause for refreshment

by Jim Butterfield
Copyright © 1989 Jim Butterfield

Jim Butterfield is a Toronto programmer and author whose articles, books, lectures and television programs have made his name a household word among Commodore users throughout the world since the early days of the KIM-1 microcomputer.

You may wonder if this column will remain stuck on DOS operations for a while yet. The answer is yes. The DOS library tends to support short programs that will allow you to explore facets of assembly language. You won't get writer's cramp, and the listings won't run on for page after page.

We will soon venture briefly into Intuition and graphics, just to show that it's accessible to us. When we do, you'll notice that the business of setting up a window, with possible gadgets and requesters, can take up a lot of code and magazine space. But, yes, it will be done to show that these areas are not forbidden ground.

You may also wonder if we will soon leap into blitters, coppers, layers, timers and other system complexities. No hurry. These are system elements common to many other languages such as C, Modula-2 and even AmigaBasic to some extent. We'll focus first on handling the code side of assembler.

This is partly a 'taking stock' column. It's time to backtrack over things we know, adding a little more detail, looking at how various development systems are used, and looking forward to future material. The objective is to make you feel more comfortable with the overall Amiga environment. The coding objectives will involve subroutine calls and some math work.

The programs we have written so far look for the *command tail*, the address of which is supplied to us in register A1. For example, if we enter the command **`rFile Zonk`**, program *rFile* will be loaded and started, and register A1 will contain a pointer to the letter 'Z' of *Zonk*. The length (in register D0) will be five characters, and behind the letter 'k' in memory there will be a newline character, *not* a binary zero.

If, in the above example, we wanted to handle a file named *Zonk*, everything would be nicely set up for us. We'd need to place a binary zero behind the letter 'k', replacing the newline. After that, we could call the DOS functions *Lock* or *Open* using the value in A1 as a pointer to the file name.

This works as expected with the standard CLI or *Shell*. But some systems – notably, early versions of the Matt Dillon shell and the debug package *MetaScope* – do things differently. For such systems, given the example above, A1 would not point to the 'Z' of *Zonk*. Instead, it would point to the space ahead of the 'Z'. If we then tried to *Open* or *Lock* using the pointer as we received it, the system would look for a file called ' Zonk', with a leading space as part of the name. Chances are, it wouldn't find it.

It's not hard to add extra code to scan past a leading space. If you think your program might be used on a system that has the extra space 'anomaly', you'll need to write a few extra 'scan-space' lines. We'll do that in the example below. A 'full-fledged' system would carefully parse the command tail, eliminate spaces, look for quotation marks, and separate various data fields. We won't be doing that for a while yet.

We have confined ourselves to single-hunk programs that start from the CLI and use simple information from the command line. We will continue to do so for a while. But in case you like to disassemble more advanced programs, I'll give you a quick rundown on this more sophisticated startup code. The description here will be narrative rather than detailed coding.

Programs can be designed to run from CLI only; that's the kind of program we have been doing here. Alternatively, programs can be designed with a completely different startup so that they run from Workbench only.

Programs can be also be designed to run from either CLI or Workbench. Such programs must contain two separate sequences of startup code: one for CLI start, one for Workbench.

A full-fledged Workbench/CLI program must, very early in its life, ask "Where did I come from?". The search for roots begins with the related question, "Who am I?". Answer: you're a *process*.

Every program started from Workbench or CLI is a process; somewhere in memory, there's a Process structure that defines and controls it. There may be many processes running at the same time within the Amiga, so the first question resolves to: "*Which* process am I?". The program asks this question by calling the Exec routine FindTask(). Find which task? The program wants to find itself, of course, so it supplies a blank name by putting a zero into register A1 before making the call (Find *me*!).

What does the program get as a result of calling FindTask()? A pointer to its own Process structure. In that structure, it can find out about itself. The first thing it will want to know is: was it called from CLI or Workbench? In the Process structure, there's an item called *pr_CLI*; if this item is non-zero, the program was called from a CLI.

Now the startup code splits. If the program was started from CLI, it follows a sequence we have seen before: opening DOS, getting input and output handles, and looking at the command tail. When the startup code has done its job, the *_main* program is called, often as a subroutine; we've done this in earlier examples.

If the program was started from Workbench, a whole different set of logic is called into operation. In this case, there will be a message coming to us from the system; we must wait for it with the Exec routine WaitPort() and then get it with GetMsg(). A message, by the way, is a set of data prepared somewhere in memory; 'getting the message' is receiving a pointer telling us where this data is located. In this case, the information in memory is called a WBStartupMsg; it is, of course, a structure. The program will make use of the contents of this structure.

The Workbench-initiated program will continue by opening the DOS library. Then it will probably want to create a window for input and output. Soon, it will ready to call the main program to do the actual work.

When the program has completed its *_main* routine, a CLI-initiated program needs to do only a few simple jobs, such as closing the DOS library. After that, the process may terminate with **RTS** (return from subroutine). A Workbench startup, in contrast, has a little more work to do. It may need to close the input/output window that was opened earlier, and it's still holding that startup message. As its last action before it quits, it must return the memory back to the system with ReplyMsg().

You may get the feeling, after having read all the above, that a Workbench startup is a fairly massive thing. Not really; most startups are 'canned', and the user just calls up the whole works by means of an **INCLUDE** statement, or using the linker. Many programmers don't know exactly what such a pre-written sequence does; they know only that it gets the job done.

In view of the size of the full startup code package, you can see why I haven't included the whole thing so far. It would hog a lot of space, and would clutter our programs with extra detail. Different assemblers and development systems call for a variety of methods to tack together bundles of code. Because of this, staying with brief CLI startup code will keep us on a common path for a while yet.

If you're anxious to create an icon-driven program right away, try using a utility such as IconX to get the effect you need.

Include files

Our last project defined a number of variable work areas by giving their values. We named the size of the FileInfoBlock structure by defining it (using an EQU, equate) as 260; we defined the position of the file size pointer within that block by giving its offset from the beginning of the structure, a value of 124.

We may continue to do things that way, but two problems arise. First, it's a lot of typing. Secondly: how do we get the actual values to be used?

Listing 1 shows a number of useful values as we might type them into our program. You might find it useful to compare **STRUCTURE** FileInfoBlock with the illustration in Figure 1. If we use these same values in many different programs, we might save time and space by putting them in a separate file (named, say, *DOS.INCLUDE*). We can then tell our assembler to include this file as part of our program with the command **INCLUDE DOS.INCLUDE**. We should try to use standard symbolic names for the values; this will make it easier for us to read other people's programs, or vice versa. Plus, standard names make it easier for us to sort through the documentation.

You could type in Listing 1 and save it as a separate file. As you went on to other jobs, you would add more include files, and expand the existing ones.

A more standard way is to use the standard Commodore 'include file' set. The files for assembly language have names that end with '.i'; these are the ones we need. The '.h' files are for use with C compilers; we would not use them with our assembler, but they are sometimes easier to read. Both .h and .i files are normal text files; you may list them to see what's there.

The Commodore include files are copyright, and are only available on disk by means of licence agreements with Commodore. If you have a 'free' assembler, chances are you won't have access to these files, and will need to look elsewhere for the information. You will find include files listed in appropriate parts of the *ROM Kernel Manual*, specifically Appendix D of the *Exec* reference. Using this document, you could type in material as you needed it.

Listing 2 shows the kind of thing you're likely to see in a Commodore include file; this one parallels part of the file *libraries/dos.i*. The coding that defines structures is readable, but contains words not normally known to the assembler: **STRUCTURE**, **LONG**, **LABEL** and **STRUCT**, for example. These words are defined elsewhere (usually in a file called

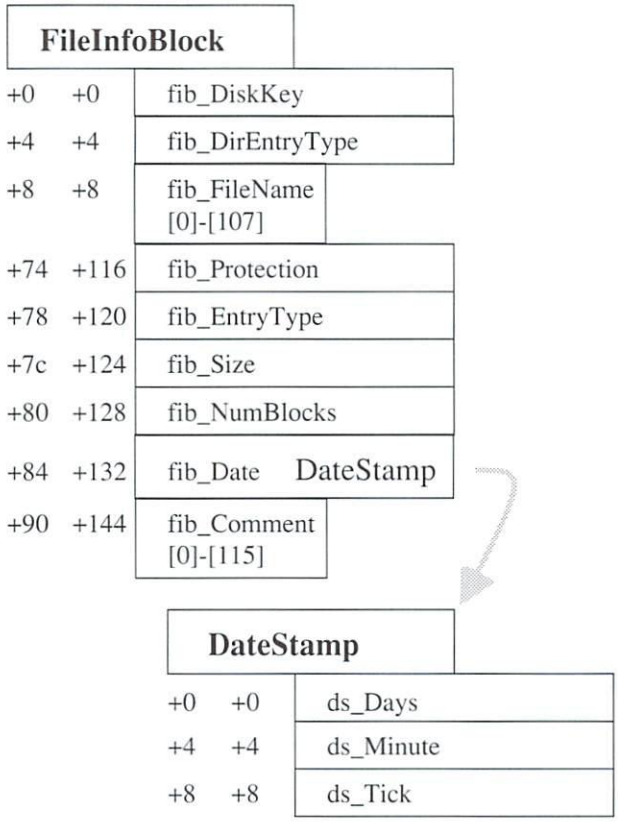


*Figure 1: the **FileInfoBlock** structure*

Listing 1. *Definitions as they might appear at the start of a program, or in a user-written "include" file. This corresponds to part of the "libraries/dos.i" file.*

```
MODE_OLDFILE          EQU   1005
ACCESS_READ           EQU   -2

; STRUCTURE DateStamp,0
ds_Days       EQU 0        ; long
ds_Minute     EQU 4        ; long
ds_Tick       EQU 8        ; long
ds_SIZEOF     EQU 12       ; size of DateStamp

; STRUCTURE FileInfoBlock,0
fib_DiskKey        EQU 0      ; long
fib_DirEntryType   EQU 4      ; long
fib_FileName       EQU 8      ; 108 bytes
fib_Protection     EQU 116    ; long
fib_EntryType      EQU 120    ; long
fib_Size           EQU 124    ; long
fib_NumBlocks EQU 128 ; long
fib_DateStamp EQU 132 ; 12 byte structure
fib_Comment    EQU 144 ; 116 bytes
fib_SIZEOF     EQU 260 ; size of FileInfoBlock
```

Listing 2. *A typical "include.i" file. This corresponds to part of the "libraries/dos.i" file.*

```
MODE_OLDFILE          EQU   1005
ACCESS_READ           EQU   -2

STRUCTURE DateStamp,0
LONG  ds_Days              ; since Jan. 1, 1978
LONG  ds_Minute
LONG  ds_Tick
LABEL ds_SIZEOF            ; DateStamp

STRUCTURE FileInfoBlock,0
LONG   fib_DiskKey
LONG   fib_DirEntryType
STRUCT fib_FileName,108 ; Null terminated.
LONG   fib_Protection   ; bit mask "-SPArwed"
LONG   fib_EntryType
LONG   fib_Size         ; bytes in file
LONG   fib_NumBlocks    ; blocks in file
STRUCT fib_DateStamp,ds_SIZEOF ; Date file
                               ; last changed
STRUCT fib_Comment,116  ; Null terminated.
LABEL  fib_SIZEOF       ; FileInfoBlock
```

exec/types.i) as macros. The commercial package *DevPac Amiga* uses a special directive, RS (Reserve Space) within its own include files to save a little macro-calling time.

Standard include files often call other include files; they, in turn, call even more. A program that works in many library areas (Intuition, Graphics, DOS, and so on) can call a staggering amount of include file material. Note that these includes generate no code; they are there solely to define symbols. An include set can easily generate several thousand symbols, of which only a dozen might be used.

The Amiga is a big machine; it has the memory capacity and programming speed to cope with all this overhead. But reading this material and looking through the extra symbol tables do increase assembly time. The amount of the slowdown depends on the particular assembler; some are more efficient than others. For this reason, some programmers prefer to write their own include files, using only the labels that they use frequently. These files would be similar to those of Listing 1.

Include files are usually called in at the beginning of your assembly code. As we have mentioned, the standard files contain only symbol definitions. You may wish to create other kinds of files that contain code, including them at the proper point within your program. The INCLUDE command will bring them in, whatever they are. There is no need to note include symbols as XREFs; they are not external. They are within your program, and the assembler will find and use them.

There are other symbols, such as the _LVO calls, that are not usually defined by means of INCLUDE. You may do so if you

wish; but the more usual way is to tell the assembler that they are XREFs (external references), so that the linker will find them for you later. The linker will find these definitions in an area called the *library*. Typically, your assembler will be furnished with two separate directories. The *include* files may be brought in to be used by your assembler. The *library* files will be checked by the linker. If you don't have the library files, you can usually make do with a freely distributable file called *SmallLib* (available on *Transactor for the Amiga* Volume 2, Issue 1 disk).

Includes and your assembler

We will write a program to calculate the size of a file. Its working name will be *GetSize*, which means you should name the source *GetSize.asm*. Depending on the assembler you use, there are various methods of hooking in with the include files. In the following quick rundown, I'll be a little vague on the exact directory path for these files. Many of us copy the include files that we use most often into a directory at a convenient location. If you can, put this on the disk that contains your assembler; this will save possible disk swaps. Users with hard disks will almost certainly park their include files there. If you have plenty of memory, copying your favourite include files into a RAM drawer will save you time if you do many assemblies in a session.

The list below is a sampling of the commands needed for startup. In some cases, I've shown extra options, such as a `-l` 'list' output file. Most assemblies will need to be followed by a link (using *Blink*) operation. This will usually be done with the command:

```
BLink GetSize.o TO GetSize LIB df0:library
```

You may choose extra options, such as map, as you wish.

Metacomco: Assemble with the command:

```
ASSEM GetSize.asm -o GetSize.o -i sys:include -l ram:list
```

C.A.P.E.68K (Inovatronics): This package likes to assemble directly from the editor menu. Set the include path by selecting menu item *Assembler Options/Include Path List*, entering something like `df0:includes`. When linking, keep in mind that the library files are in a drawer called *LinkFiles*.

DevPac Amiga (HiSoft/MichTron): *Devpac* wants to assemble directly from the editor menu, although assemblies can be called directly from the CLI. If library information is not needed, you may skip the linker step; if you plan to assemble and link in separate steps, be sure to select this on the assembler menu, and keep in mind that you *must* direct the resulting object to a disk file (this can, of course, be the RAM disk). If you use the editor to assemble, you will need to add information on the include path to your source program: something like `INCDIR df0:include/` will do the job. Do not forget that slash character at the end; it's important. The system supplies two sets of include files: its own, and those of Commodore.

Lattice: Assemble with the command

```
asm -idf1:Assembler_Headers/ GetSize.asm
```

The options (such as `-i`) must come first. There must be no space following the `-i`; note the underscore character in `Assembler_Headers`, and the following slash character. You might wish to rename the drawer to something easier to type. The Lattice assembler seems to be disoriented by include files, and often needs a `csect` statement at the beginning of the program code to make things happy.

ASM68010 (Compute! Books): The include files that come with this system use non-standard names. This may make it difficult to use code supplied from other systems. For example, the standard label **_LVOOpenLibrary** is replaced by LVO.OPENLIBRARY – close, and quite readable, but not fully compatible. Assemble using the command:

```
ASM GetSize.asm -o GetSize.o
```

and link in the usual way. Or, if library information is not needed, issuing the command:

```
ASM -A GetSize.asm -o Getsize
```

will result in an executable program rather than an object file.

AssemPro (Abacus/Data Becker): Include files are supplied; a full-path directory name must be specified with a source file INCLUDE statement. AssemPro does not use a linker, but generates executable files directly; this means you can't use XREF and must define your library entries explicitly. The demo files do this mostly with EQU statements; such statements could easily be put onto a separate file and called up with INCLUDE.

The manual suggests that you define such labels and then save them to a special file with SLABEL, calling them back with the directive ILABEL.

A68K (Charlie Gibbs): No include files come with this free assembler. If you have them, or have typed in the equivalent files, use the command:

```
A68K GetSize.asm -oGetSize.o -isys:include -lram:list
```

Note that there must be no spaces after flags such as `-i` or `-o`.

Asm68K (Wes Howe): Again, this free program does not furnish include files. If you have them, be sure to use the *include.strip* version: as the documentation notes, the 'full' include files contain end-of-line comments marked with an asterisk that could be confused with the multiplication symbol. Use the command:

```
Asm68K GetSize.asm -i include: -l ram:list
```

Note that spaces are permitted after flags such as `-i`.

Some arithmetic

If you have a numeric value that you want to print, assembly language requires you to change the binary information to a stream of characters, ready for printing. This takes a little math; nothing very hard.

You have a choice of methods. You could create characters starting from the left. This involves dividing successively by 10000, 1000, 100 and so on, or a similar procedure. This method lets you output the characters as they are produced, if you wish. There are methods that generate all the digits at the same time. Most of these involve using the **ABCD** (Add, Binary Coded Decimal) instruction. We won't use these here.

Finally, the method we *will* use. You can create the digits starting at the low end of the number. This involves dividing by ten repeatedly; after each division, the remainder is the desired digit. For example, if we had decimal 456: divide by 10 gives 45, remainder 6; the 6 is our last digit. Divide 45 by 10 gives 4, reminder 5; the 5 is our previous digit (giving digits 5 and 6 so far). Finally, dividing 4 by ten yields 0, with a remainder of 4 that becomes our first digit.

This method is quite easy on the 68000, since **DIVU** (divide unsigned) is an instruction. **DIVU** gives us a remainder value as part of the calculation. The 68000 does division relatively slowly, taking about 25 times longer than for an addition. For this reason, programming lore says 'avoid divide and multiply instructions if possible'. The operations become much faster in the 68020 and 68030; in the meantime, **DIVU** will do a neat and relatively efficient job for us here.

We will use a simple method with only one division in the loop. Because the **DIVU** instruction gives a result of only 16 bits, this method won't work on big numbers; but it will do well enough in this program.

The *GetSize* program

The code starts with the include files we need.

```
; csect code                  ; for lattice asm
; INCDIR df0:include/         ; for DevPac Amiga
  INCLUDE "exec/types.i"
; above not needed if next line invokes types.i
  INCLUDE "libraries/dos.i"
;
; Exec library calls
  xref _LVOAllocMem           ;-$C6
  xref _LVOFreeMem            ;-$D2
  xref _LVOCloseLibrary       ;-$19E
  xref _LVOOpenLibrary        ;-$228
; DOS library calls
  xref _LVOWrite              ;-$30
  xref _LVOOutput             ;-$3C
  xref _LVOLock               ;-$54
  xref _LVOUnLock             ;-$5A
  xref _LVOExamine            ;-$66
; define work frame
; Working variables in frame A4
  CmdTail      EQU -$4
  DosBase      EQU -$8
  OutHandle    EQU -$C
  FileLock     EQU -$10
  AllocBlock   EQU -$14
;
;-- Initial setup:
Startup   link  a4,#-$14
```

We have allocated a frame to hold our working variables. The variables themselves can be seen in the **EQU** statements above. We now scan for leading spaces. They should not be there; but, as mentioned earlier, some systems put them in.

```
          move.b #$0,-1(a0,d0.w) ; insert binary zero
SpaceLoop move.b (a0)+,d0        ; examine character
          cmp.b #$20,d0          ; is it space?
          beq.s SpaceLoop        ; yes, look again
          subq.l #1,a0           ; no, back up
```

There are many ways of trimming back an address pointer. For values of eight or less, you can't beat the **SUBQ** instruction. For larger backups, a form similar to **LEA -1(A0),A0** does the job in a clever way; can you see how it would work?

```
          move.l a0,CmdTail(a4) ; Pointer to command tail
```

The **DOS** library is opened in the usual way, and if all is successful we call our main job as a subroutine, *MainJob*

```
          lea dosname(pc),a1    ; Name 'dos.library'
          clr.l d0              ; Any version (0).
          move.l $4,a6          ; Using Exec library,
          jsr _LVOOpenLibrary(a6) ; open Dos library;
          move.l d0,DosBase(a4) ; remember DosBase ptr.
          beq.s StartupQuit     ; Dos didn't open!
          bsr.s MainJob
          move.l DosBase(a4),a1 ; DosBase to a1;
          move.l $4,a6    ; then using Exec library,
          jsr _LVOCloseLibrary(a6) ; close Dos library.
StartupQuit unlk a4
          rts                   ; End of Program
;
;-- Get CLI outhandle:
MainJob   move.l DosBase(a4),a6 ; Set Dos library
          jsr _LVOOutput(a6)    ; get CLI outhandle,
          move.l d0,OutHandle(a4) ; and remember it.
```

After picking up the output filehandle, the program gets a lock on the file and allocates memory to hold the FileInfoBlock. Then it's time to get the file data using the **DOS** Examine function.

```
; get a 'lock' on the file, if possible.
          move.l CmdTail(a4),d1
          moveq #ACCESS_READ,D2   ; Get a read lock
```

```
        jsr _LVOLock(a6)
        move.l d0,FileLock(a4) ; and store the
                               ; lock pointer.
        beq  DosQuit           ; couldn't get lock
; ask for memory to hold a FileInfoBlock.
        move.l $4,a6      ; this calls for Exec
        moveq #0,d1       ; any type of memory
        move.l #fib_SIZEOF,d0 ; size of block needed
        jsr _LVOAllocMem(a6)      ;  go get it
        move.l d0,AllocBlock(a4) ; save the
                                 ; block address
        beq DropLock           ; quit if no memory
; Now get information on the file
        move.l DosBase(a4),a6     ; set Dos library
        move.l FileLock(a4),d1    ; lock
        move.l d0,d2      ; file info block address
        jsr _LVOExamine(a6)       ; go for the data
; Assume Examine never fails. Go for the size.
        move.l d2,a2      ; file info block address
        move.l fib_Size(a2),d0    ;file size
```

The size of the file is now in register D0, and the program is ready to convert it to decimal and print it. The next part is treated as if it were to become a standard subroutine: the coding is careful not to disturb any registers other than D0, D1, A0 and A1.

The program does this by copying registers to the stack; that's the **MOVEM** (move multiple registers). The registers are restored at the end of the operation with another **MOVEM**.

This register-preserving is an exercise; it's not needed in this program. You can see how it's done; in this case, we save three registers. Even then, we're guilty of overkill; register A2 will be restored to its original value after the **LINK/UNLK** sequence.

You might plan to make a piece of code such as this into a subroutine for later including, linking, or macro-calling. When you do, it's wise to follow the Amiga system practice of preserving the contents of all registers numbered '2' or higher. Make it a habit, and you won't need to worry about which registers are safe and which are not.

The code starts by creating a character work area (using **LINK**). The area is cleared to ASCII spaces.

```
; This decimal conversion subroutine works correctly
; ONLY with word values (65535 and lower). A more
; elaborate routine is needed for longword
; conversion (BIG files).
        movem.l a2/d2-d3,-(a7)
        link a2,#-8
; clear output area to spaces
        moveq #7,d1
        lea -8(a2),a0
ClearDig  move.b #$20,(a0)+
        dbf d1,ClearDig
        move.b #$0a,-(a0)
```

The last character was replaced by a newline character. Since we're working from the back of the string, the newline will be behind the numeric string; we'll print it. After the number is printed, we'll go to a fresh line. Now divide to get the next digit.

```
; Value is in register D0
; pull out digits from low end
NextDig   divu #10,d0
```

After the division, the quotient is in the lower half, or word, of register D0, and the remainder is in the upper half. We want to work with the remainder first, so we swap the two halves.

```
        swap d0
```

Since we have divided by ten, the remainder must be in the range zero to nine. It's a binary value, located in the low byte of register D0; we will change it to an ASCII character by adding a couple of 'zone bits' using the **OR** command. Then we store it in the buffer area, working downward as we go.

```
        ori.b #$30,d0
        move.b d0,-(a0)
```

The character has been put away. Clear the lower half of the register, and then swap the quotient (still in the upper word) to the low end. If the quotient is zero, we have done our conversion.

```
        move.w #0,d0
        swap d0
        bne.s NextDig
```

The conversion is complete. By the way, the program comment (regarding the range of values that will work) is not precisely correct. This routine cannot convert full 32-bit values to ASCII, but it will work successfully on higher values than the stated 65535. Can you identify its true upper limit?

```
; print the eight bytes
        move.l OutHandle(a4),d1
        lea -8(a2),a0        ;start address
        move.l a0,d2
        moveq #8,d3        ;length
        jsr _LVOWrite(a6)
        unlk a2
        movem.l (a7)+,a2/d2-d3
```

The number has been printed, complete with leading spaces (if any) and the following newline. There's nothing left to do but clean up. Return the allocated block, remove the lock, and return.

```
FreeInfo  move.l $4,a6      ; set Exec library
          move.l AllocBlock(a4),a1 ; memory block
                                   ; address
          move.l #fib_SIZEOF,d0    ; size of block
          jsr _LVOFreeMem(a6) ; release this memory
DropLock  move.l DosBase(a4),a6    ; set Dos Library
```

```
            move.l FileLock(a4),d1 ; take off the lock
            jsr_LVOUnLock(a6)
DosQuit     rts                          ; exit program.
dosname     dc.b 'dos.library',0
            end
```

GetSize won't dazzle your friends; it's not a profound program, but it has opened up a few new areas for us. We've had a chance to play with the INCLUDE operation; we've worked at preserving some register contents; and we've done an efficient binary-to-decimal number conversion.

Listing: *GetSize.asm*

```
;
;   Show File Size - Jim Butterfield.  April 29/89.
;
; Metacomco - ASSEM GetSize.asm -o GetSize.o -i sys:include -l ram:list
; C.A.P.E - menu "Assembler options/Include Path List" dfx:includes
; a68k (gibbs) - A68K GetSize.asm -oGetSize.o -isys:include -lram:list
;      note  charlie doesn't like spaces after say -i
; Lattice:  asm -idf1:Assembler_Headers/ GetSize.asm.. note the slash
;         all options must be BEFORE filename
;         -l to 'standard output' - redirection needed
;         lattice gives NO completion advice
; DevPac: GenIm GetSize.asm -idf0:include  or from menu, as below
;
; csect code                    ; for lattice asm
 INCDIR df0:include/            ; for DevPac
 INCLUDE "exec/types.i"
; above not needed if next line invokes types.i
 INCLUDE "libraries/dos.i"
;
; Exec library calls
            xref _LVOAllocMem          ;-$C6
            xref _LVOFreeMem           ;-$D2
            xref _LVOSetSignal         ;-$132
            xref _LVOCloseLibrary      ;-$19E
            xref _LVOOpenLibrary       ;-$228
; DOS library calls
            xref _LVOOpen              ;-$1E
            xref _LVOClose             ;-$24
            xref _LVORead              ;-$2A
            xref _LVOWrite             ;-$30
            xref _LVOOutput            ;-$3C
            xref _LVOLock              ;-$54
            xref _LVOUnLock            ;-$5A
            xref _LVOExamine           ;-$66
; define work frame
; Working variables in frame A4
CmdTail     EQU  -$4
DosBase     EQU  -$8
OutHandle   EQU  -$C
FileLock    EQU  -$10
AllocBlock  EQU  -$14
;
;-- Initial setup:
Startup     link   a4,#-$14
            move.b #$0,-1(a0,d0.w)   ; insert binary zero
SpaceLoop   move.b (a0)+,d0          ; examine character
            cmp.b  #$20,d0           ; is it space?
            beq.s  SpaceLoop         ; yes, look again
            subq.l #1,a0             ; no, back up
            move.l a0,CmdTail(a4)    ; Remember ptr to argument line.
            lea    dosname(pc),a1    ; Name 'dos.library'.
            clr.l  d0                ;   Any version (0)
            move.l $4,a6             ;   Using Exec library
            jsr    _LVOOpenLibrary(a6)  ;   Open Dos library.
            move.l d0,DosBase(a4)    ; Remember DosBase ptr.
            beq.s  StartupQuit       ;      dos not opened
            bsr.s  MainJob
            move.l DosBase(a4),a1        ; DosBase to a1;
            move.l $4,a6             ;  then using Exec library,
            jsr    _LVOCloseLibrary(a6)    ;   close Dos library.
StartupQuit unlk   a4
            rts                      ; End of Program
;
;-- Get CLI outhandle:
MainJob     move.l DosBase(a4),a6    ; set Dos library
            jsr    _LVOOutput(a6)    ;   get CLI outhandle,
            move.l d0,OutHandle(a4)  ;   & then remember it.
;
; get a 'lock' on the file, if possible.
            move.l CmdTail(a4),d1
            moveq  #ACCESS_READ,D2   ; get a read lock
            jsr _LVOLock(a6)
            move.l d0,FileLock(a4)   ; store the lock pointer
            beq    DosQuit           ; couldn't get lock
; ask for memory to hold a FileInfoBlock.
            move.l $4,a6             ; this calls for Exec
            moveq  #0,d1             ; any type of memory
            move.l #fib_SIZEOF,d0    ; size of block needed
            jsr    _LVOAllocMem(a6)  ; go get it
            move.l d0,AllocBlock(a4) ; save the block address
            beq    DropLock          ; quit if no memory
; Now get information on the file
            move.l DosBase(a4),a6    ; set Dos library
            move.l FileLock(a4),d1   ; lock
            move.l d0,d2             ; file info block address
            jsr _LVOExamine(a6)      ; go for the data
; Assume Examine never fails.  Go for the size.
            move.l d2,a2             ; file info block address
            move.l fib_Size(a2),d0   ;file size
; This decimal conversion subroutine works correctly ONLY with
; word values (65535 and lower).  A more elaborate routine is
; needed for longword conversion (BIG files).
            movem.l a2/d2-d3,-(a7)
            link  a2,#-8
; clear output area to spaces
            moveq #7,d1
            lea   -8(a2),a0
ClearDig    move.b #$20,(a0)+
            dbf    d1,ClearDig
            move.b #$0a,-(a0)
; pull out digits from low end
NextDig     divu  #10,d0
            swap  d0
            ori.b #$30,d0
            move.b d0,-(a0)
            move.w #0,d0
            swap d0
            bne.s  NextDig
; print the eight bytes
            move.l OutHandle(a4),d1
            lea    -8(a2),a0         ;start address
            move.l a0,d2
            moveq  #8,d3             ;length
            jsr    _LVOWrite(a6)
            unlk a2
            movem.l (a7)+,a2/d2-d3
FreeInfo    move.l $4,a6             ; set Exec library
            move.l AllocBlock(a4),a1 ; memory block address
            move.l #fib_SIZEOF,d0    ; size of block
            jsr    _LVOFreeMem(a6)   ; release this memory
DropLock    move.l DosBase(a4),a6    ; set Dos Library
            move.l FileLock(a4),d1   ; take off the lock
            jsr    _LVOUnLock(a6)
DosQuit     rts                      ;   exit program.

dosname     dc.b   'dos.library',0
   end
```

The Heart Of Exec: Interrupts

The core of the operating system, explained by its author

by Carl Sassenrath

Carl Sassenrath is an Amiga Guru. As one of the original designers of the Amiga software system, Carl is the master architect of the Amiga's Multitasking Operating System and the creator of the Exec Library. Since receiving his Electrical Engineering degree in 1979, Carl has been involved in the design of advanced operating systems for Hewlett Packard, Commodore-Amiga, and Apple Computer. More recently, Carl heads his own software development company, Sassenrath Research. He is also the author and publisher of the ***Guru's Guide****, a detailed technical journal about the Amiga.*

Introduction

Buried deep within the Amiga ROM Kernel it lies: a small, inconspicuous module of code. Yet through this tiny module flows the life blood of all Amiga programs. Steady and sure, it beats to the rhythm of all activities: every mouse movement, every key press, every vertical blanking blink of an eye; an audio chirp, a disk DMA, the time of day slowly ticking. It is the heart of multitasking, the foundation of Exec: it is the Amiga Interrupt System.

The Interrupt System is the oldest module in the ROM kernel. Developed in late 1983, the interrupt system was designed for both flexibility and performance. It is an Exec Library submodule consisting of approximately 1K bytes of fine-tuned assembly code.

The Interrupt System is primarily responsible for synchronizing most of the run-time activities within the busy Amiga. Each of these activities occurs on demand, at its own speed, without any knowledge of the other activities that share the same moment in time. Yet, all of these activities must bid for the attention of their controlling software executing on the 68000 microprocessor.

Hardware

The hardware of the Amiga can generate interrupts for video, audio, disk, blitter, coprocessor, keyboard, serial port, parallel port, timers, and several external devices. All of these interrupts must pass through special control circuitry within the Paula custom chip in order to capture the attention of the 68000 microprocessor. This circuitry, called the *Amiga Interrupt Controller* (AIC), provides the basic functions needed to enable, request, service, and clear interrupts. A total of fourteen interrupt sources feed into the AIC from from the various hardware devices on the Amiga:

Video — Each time the video beam returns to the top of the screen, a vertical blanking interrupt occurs.

Copper — The Amiga coprocessor (Copper) has the ability to cause an interrupt for several conditions.

Blitter — The Amiga blitter can signal that it has finished a request with an interrupt.

Audio — Each of the four Amiga sound channels can cause its own interrupt.

Disk — The floppy disks cause two types of interrupts: one when a data transfer is complete, the other when a particular bit pattern has been read.

Serial — The serial port can generate interrupts on two conditions: when it has received a new character, and when it has finished sending a character.

Ports — Special external devices also generate interrupts. These include the keyboard, the parallel port, network serial port, and several timers.

Interrupts are not fed to the 68000 CPU directly; they pass through the AIC first. The software controls and monitors the AIC through a set of custom chip registers. Four of these registers are used to communicate with the AIC. Two registers are used for reading, two are used for writing:

`INTENA` to enable and disable interrupts (write only)

`INTENAR` to read the enable register (read only)

`INTREQ` to request and clear interrupts (write only)

`INTREQR` to read the request register (read only)

Each of these registers is a word in size (16 bits). The lower fourteen bits of each register are identical – one bit for each interrupt source. By changing these bits, each interrupt can be separately enabled, disabled, requested and cleared.

To use the AIC, your code must read and write these registers. The system does not provide functions to access these registers, so your code will change them directly. When doing this, *take care never to read one of the write only registers.* This would cause a random value to be written into the register.

To access these registers, you will need to include two special system files within your program:

custom specifies the offsets for custom chip hardware registers
intbits defines the interrupt bits for the AIC registers

The first file gives you access to the AIC registers. This makes it relatively easy to access the registers from both C and Assembly code. For example, you can use these structure members to access the interrupt registers:

custom.intena
custom.intenar
custom.intreq
custom.intreqr

Assuming that you have defined **custom** as something along the lines of

```
#define custom (*((struct Custom *) 0xdff000))
```

The second include file (*intbits*) contains a means of referencing a particular interrupt bit. Here is a list of the interrupts by bit number:

15	INTB_SETCLR	- set and clear the bit mask
14	INTB_INTEN	- master interrupt (enable only)
13	INTB_EXTER	- CIA-B + external (high pri)
12	INTB_DSKSYNC	- disk word synchronized
11	INTB_RBF	- serial, receive buffer full
10	INTB_AUD3	- audio channel 3 ready
9	INTB_AUD2	- audio channel 2 ready
8	INTB_AUD1	- audio channel 1 ready
7	INTB_AUD0	- audio channel 0 ready
6	INTB_BLIT	- blitter finished
5	INTB_VERTB	- start of vertical blank
4	INTB_COPER	- coprocessor
3	INTB_PORTS	- CIA-A + external (low pri)
2	INTB_SOFTINT	- software interrupt
1	INTB_DSKBLK	- disk block done
0	INTB_TBE	- serial, transmitter buffer empty

This file also contains another set of notations for the same bits using the prefix INTF. These use a 16 bit word as a value rather than a particular bit number. You will use both notations depending on what it is you need to do.

A special technique is used to modify bits in the INTENA and INTREQ registers. The most significant bit (bit 15) in these registers is called the SETCLR bit. It is used in conjunction with a 14 bit mask to change individual bits in the register. When the SETCLR bit is one, all one bits in the mask will be set. When SETCLR is zero, all one bits in the mask will be cleared. With this mechanism, several bits can be changed at the same time without affecting the other bits.

Using this method of changing bits, you can independently enable, disable, request, and clear interrupts. You do not need to know the status of the other interrupts to make a change to a particular interrupt. This makes interrupt enabling and disabling almost trivial, and saves the system considerable overhead.

For example, in C the statements:

```
custom.intena = INTF_SETCLR | INTF_AUD3;

custom.intena = INTF_AUD3;

custom.intreq = INTF_SETCLR | INTF_AUD3;

custom.intreq = INTF_AUD3;
```

Would enable, disable, set (which the hardware normally does for you) and clear the audio interrupt for sound channel 3. You could also test the status of the channel with:

```
if (custom.intenar & INTF_AUD3) {...}

if (custom.intreqr & INTF_AUD3) {...}
```

The first would test if the AUD3 interrupt was enabled, the second would determine if it was set.

One other bit deserves special attention. INTEN (bit 14) of the INTENA register is the master enable for all interrupts. When set to zero it disables all interrupts. When set to one it enables all interrupts that have their enable bits set. *This is the one INTENA bit that programs are not permitted to change directly.* This bit is used by the system software to disable interrupts during critical sections. If your code modified the master enable, it might inadvertently turn interrupts back on at the wrong time. This could create serious, intermittent reliability problems for the system software.

Two Exec Library functions are provided to solve this problem. The Disable() function shuts off all interrupts; Enable() turns them back on. These functions are used in rare cases for short sections of code when you don't want any interrupts to occur. What's special about them is that they do the right thing when nested. For example, if Disable() is called three times, then Enable() must be called three times before interrupts actually become enabled again. It turns out that many of the Exec Library system functions use Disable() and Enable(), so nesting may be occurring even if you don't know about it. In

the end, these functions prevent interrupts from becoming enabled within critical sections of system code. They should be used only with great skill and care.

Exec: handlers and servers

Whenever the 68000 microprocessor receives an interrupt, control is first passed to the Exec part of the ROM Kernel. Control is never passed directly from the interrupt to a program's interrupt handling code. There are two main reasons for this: some of the interrupts may be shared by more than one program; and the system must evaluate the state of multitasking at the end of each interrupt.

Once Exec receives control of an interrupt, it deals with it as a Server or as a Handler. Servers provide a simple means of sharing interrupts. Exec supports servers for several interrupts: vertical blank, coprocessor, and all external ports. Handlers provide a direct connection to interrupts that cannot be shared. Handlers usually interact with the AIC and thus require more attention to details than a typical server. Handlers are used to process interrupts for the serial port, floppy disk, audio channels, and blitter.

The choice as to whether your code is a server or a handler will depend on the interrupt you intend to process. Both styles have the identical structure, but involve different system functions and AIC interactions.

Interrupt structure

To program interrupts, there is only one data structure of importance: the Interrupt. Handlers and servers both use this same structure. The definition for this structure can be found in the Exec include file: *interrupts.h*. The C code definition is:

```
struct Interrupt
{
    struct Node is_Node;
    APTR is_Data;
    VOID (*is_Code)();
}
```

As you can see, this structure begins with a standard Exec list node. This node is used to attach the structure to the appropriate system list. Of importance within the node are the fields for priority, type, and name. All these fields are used with interrupts.

The priority field determines where your interrupt will be placed within a server chain. Higher priorities will always be serviced first. The name field is used to identify the interrupt. It points to a zero terminated ASCII string of characters. You will want to give a name to your interrupt so it can be identified by other programs. The best names are those similar to the name of the parent program, for example:

"WizGraph.library"
"SuperEdit.program"
"FastDisk.device"

The type field is used to indicate the Exec structure type. A numeric constant (from *nodes.h)* NT_INTERRUPT should be used for all types of interrupts.

The is_Data field is used to point to a data area to be used when the interrupt occurs. This is normally used for ROM code (in which there can be no fixed RAM addresses), but it is also a handy place to store a pointer to any relevant data.

The is_Code field of the interrupt structure is a pointer to the entry point of an interrupt processing routine. When the interrupt occurs, this routine is called by Exec as if it were a subroutine.

Functions

Interrupts are supported with several Exec library functions. These functions primarily provide a means of connecting and disconnecting interrupts to the system:

SetIntVector	establish a new interrupt handler
AddIntServer	add a new interrupt server
RemIntServer	remove an interrupt server
Cause	generate a software interrupt

It is beyond the scope of this article to describe these in more detail, but they are documented in detail in the *Guru's Guide, Meditation #1*.

A simple server example

As an example of the simplest type of server interrupt, here is a program that does nothing but count vertical blanking intervals:

```
/****************************/
/***  Simple Interrupt    ***/
/****************************/

#include <exec/interrupts.h>
#include <hardware/intbits.h>

VOID    IntrProc(); /* forward to interrupt routine */
long    IntrCount;  /* vertical blank counter    */

/* Initialized interrupt structure */
struct Interrupt VertBlank =
{
   NULL,                    /* node pred*/
   NULL,                    /* node succ*/
   -60,                     /* node pri */
```

```
    NT_INTERRUPT,           /* node type*/
    "simple.example",       /* node name*/
    (APTR)&IntrCount,       /* data ptr */
    &IntrProc               /* code ptr */
};

#asm    /*  Interrupt Server Routine */
_IntrProc:
      addq.l  #1,(a1) ; increment IntrCount
        moveq   #0,d0   ; Z-flag - do next server
        rts
#endasm

main()
{
    extern int Enable_Abort;

    Enable_Abort = 0;   /* prevent a CTRL-C */
    IntrCount = 0;
    AddIntServer(INTB_VERTB, &VertBlank);
    while (IntrCount < 200) printf("%d\n",IntrCount);
    RemIntServer(INTB_VERTB, &VertBlank);
}
```

This program was compiled with Manx Aztec C 3.6A using the +L flag to force 32 bit integers (it should be linked with Aztec 'C32' library). Let's examine it in closer detail.

The first declaration is a forward reference to the Interrupt server routine. This is needed to keep the compiler happy within the Interrupt Structure definition.

The second declaration defines a variable that will be used to hold the number of vertical blanking intervals. This variable will be referenced from the interrupt code in an indirect fashion (to be described shortly).

Next, notice that the Interrupt structure is initialized statically within the program. This is done to simplify the example, but there is really no reason why you couldn't allocate the structure and initialize it within the code.

The structure contains a node priority of -60. This very low priority is assigned to keep the program from interfering with other, more important vertical blanking interrupts that will be processed in the same chain.

The name 'simple.example' has been given to the interrupt node for easy identification from debuggers and interrupt monitors.

Finally, the data and code pointers are established. The data pointer refers to the address of the variable that holds the number of vertical blanking intervals. The code pointer points to the interrupt server routine.

The interrupt server routine is a simple, short, and fast assembly code subroutine. It will be called by Exec from supervisor mode when the interrupt occurs. Because it is being called within supervisor mode, it should be short and to the point. There are many restrictions that apply to operation within supervisor mode on the Amiga. The biggest restriction being that multitasking does not operate from supervisor mode: you cannot call many of the normal system functions. You should not, for example, attempt to perform any type of device related I/O.

Your assembly code function will be called within the context of the Exec system library. Exec saves several of the CPU registers before it dispatches your server. The standard scratch registers D0, D1, A0 and A1 are preserved by Exec, so your code can operate without needing to save them. (Note: there is a bug in the Graphics Library which will require that you preserve the value of A0 if you're using a vertical blank server with a priority greater than nine). In addition, A5 and A6 are often used by servers, so they are also preserved by the system.

Two registers are passed to your code with special values. The A1 register holds a pointer to data as stored in the is_Data field of your interrupt structure. The A5 register holds a pointer to your code's entry point (the is_Code field).

Because this server routine is so simple, it does not need to reference values in the Manx compiler's data segment; If you needed to access other global variables, you would need to set up the A4 register appropriately.

In the main function of the program, the AddIntServer() function is used to link the initialized interrupt structure into the system. The Manx Enable_Abort flag is used to prevent a CTRL-C abort while the program is running. This is very important. If you were to stop the program by pressing a CTRL-C, the program would exit without removing the interrupt server. This would cause the system to crash as as a result of your program's code and data segments being freed and later overwritten in memory.

A more complex example

As a more complex interrupt example here is a program that signals a task after a specified number of vertical blanking intervals.

```
/*********************************/
/***  Task Signal Interrupt  ***/
/*********************************/

#include <exec/exec.h>
#include <hardware/intbits.h>


long IntrCount = 0;  /* vert-blank counter */
long ASignal = -1;   /* task signal to use */
struct Task *ATask;  /* task to signal   */
```

```
/* Primary Interrupt Code: */
DoInterrupt()
{
   if (--IntrCount == 0) Signal(ATask,1 << ASignal);
}


/* Interrupt Processing Code */
VOID IntrProc();
#asm
   PUBLIC   _int_start,_int_end
_IntrProc:
   move.l   a0,-(sp) ; for gfx lib bug
   jsr   _int_start  ; MANX support
   jsr   _DoInterrupt
   jsr   _int_end ; MANX support
   move.l   (sp)+,a0
   moveq.l  #0,d0    ; do next server
   rts
#endasm


/* Initialized interrupt structure */
struct Interrupt VertBlank =
{
   NULL,        /* node pred*/
   NULL,        /* node succ*/
   -60,         /* node pri */
   NT_INTERRUPT,      /* node type*/
   "simple.example", /* node name*/
   0,        /* data ptr */
   &IntrProc       /* code ptr */
};


main()
{
   int count;
   extern long AllocSignal();
   extern struct Task *FindTask();

   ASignal = AllocSignal(-1);
   if (ASignal == -1)
      exit(100); /* could not allocate signal */
   ATask = FindTask(NULL);
   AddIntServer(INTB_VERTB, &VertBlank);

   for (count = 0; count < 10; count++)
   {
      IntrCount = 60;
      Wait(1 << ASignal);
      printf("%d\n", count);
   }

   RemIntServer(INTB_VERTB, &VertBlank);
   FreeSignal(ASignal);
}

_abort() /* Replaces MANX ^C termination */
{
   RemIntServer(INTB_VERTB, &VertBlank);
   FreeSignal(ASignal);
   puts("^C");
   exit(20);
}
```

This program is designed to be more generally useful than the previous example. It can be easily expanded to do many other types of operations.

Notice that the program makes use of a task signal to inform the parent task that it is time to resume operation. This is a convenient mechanism for informing a task that interrupt code has finished. More details about tasks are available from the ROM Kernel Manual.

The IntrProc() routine only deals with setting up the C environment and returning properly. It calls the DoInterrupt() function to do the real work. This style of coding is used for two reasons: to save and restore the A0 register, to setup the C compiler's data segment (A4), to save the compiler's scratch registers, and to make sure that the code returns with the CPU's zero flag set. If we programmed it entirely in C rather than assembly code, we could not guarantee that A0 would be saved nor that the CPU's zero flag would be set.

This example also properly handles CTRL-C by redefining the Manx _abort() function. A CTRL-C will cause the program to terminate in a safe manner.

Conclusion

This article has described the basic structure and operation of interrupts on the Amiga. It has given a few simple examples that should be of use for programmers who wish to experiment with interrupts. There is, however, a lot more to the interrupt system than can be described in just one magazine article. If you are planning to create a program that requires direct interrupt control, it would be wise to seek further knowledge. Within the unprotected multitasking environment of the Amiga, one flawed program can bring all the others down.

A complete and detailed treatment of the Amiga Interrupt System is available from the Guru's Guide, Meditation #1. This 105 page softbound book provides not only a theoretical foundation, but supplies many working examples.

A Vile of Illiteracies

Our curmudgeon slays an ass with its own jawbone

by Dick Barnes
Copyright © 1989, Dick Barnes

Dick Barnes has written and edited for 30 years. He was an engineer in R&D and backed into computing in the late 1950s as part of his work. After he retired, Dick edited and published the late, lamented ***Amigan Apprentice & Journeyman****.*

In the February 13 issue of *InfoWorld*, columnist Jeff Angus says the computer industry mutilates the English language by "doing to [it] what Hitler did to Poland in 1939". That's unfair to Adolf. The unceasing massacre of English is more like the butcheries of Attila the Hun, who slaughtered and raped in barbaric ignorance of any other way to behave.

Barbaric ignorance is the key. Two generations suckled on the boobtube have grown to adulthood in barbarous ignorance of how to read or write. Most lower schools fail to teach English – and grant diplomas to illiterates. If you don't believe that, consult the nearest university – where most high school graduates must take corrective cram courses in English before they can comprehend their textbooks and classwork.

Few study the humanities, where they might stumble across good writing. (A bachelor's degree in the humanities entitles you to starve to death after you graduate.) Most students specialize. Thereafter, the only books they read from cover to cover are textbooks in a specialty – business administration, computer science, and so on – crammed with the jargon of that trade. Result: Most university graduates write with the grace and clarity of Ooog, the caveman.

The deadly trap: homonyms

Most boobtubers know the meaning of sounds heard on the tube, but not how to spell them. They exchange homonyms (words of the same sound but of different meaning and spelling, as with 'breach' and 'breech') with hilarious results. Here are a few disasters from my collection of homonyms:

- "The horse knickered, galloped up to me, and nayed." Ogden Nash said he'd never seen a purple cow and hoped he'd never see one. I feel the same way about horses in knickers naying. The same horse was then hitched to "a one-horse open slay."
- "She waited with baited breath." Is a mint julep good bait?
- "The corporation completed faze three of its reorganization."
- "I'm a died-in-the-wool fan." Risen from the dead?
- "The general poured over the map" Tired of poring?
- "A hoard of natives attacked the village." How many natives in a hoard? A horde?
- "I fell in love with Amiga and we were united in wholly matrimony."
- "He was wounded in a navel engagement with a Nazi E-Boat off Calais." Gay hand-to-hand naval combat, no doubt.

Because you can't comprehend the next section if blinded by tears of hysterical laughter, I've reserved the worst homonymal bloopers for the conclusion.

The language changes – for the worse

I'm no reactionary purist – one who faints at the sight of a split infinitive. English changes, often for the better. The word 'mob', for example, was derived from the Latin *mobile vulgus*, a disorderly rabble. Jonathan Swift, author of Gulliver's Travels, objected to 'mob' as a barbarism. Swift was wrong. English is richer for it.

But most of the changes I read now don't enrich the language. A good new word must be easy to pronounce, crisp, short and clear. It must plug a gap in the language, as do 'boot' and 'guru' and 'nuke'. Writers can create new words by changing the old. Could we survive without the relatively new words 'sterilize', 'debunk' or 'bulldoze'? Yet most current new words are duds.

Examine 'debuted' as a synonym for 'first introduced'. It fills no hole; when pronounced properly (*day-bood*; the 't' is silent), it could mean 'booed by day'. And because Anglo-Saxons often mispronounce French words, 'debuted' is more often pronounced either as *de-bood* ('the boos are removed') or as *de-byuted*, which means you removed a butte from the landscape. If spoken as *de-butted*, it means that you cut off someone's butt. When a new word can't be pronounced with clear meaning, trash it!

To use 'authoring system' to mean a program that writes programs fills no gap and rapes the word 'author', which means a creator – the maker of anything. Could an 'authoring system' sculpt Michaelangelo's David, create Fats Waller's jazz, or write the novels of Tom Wolfe and Hemingway? Nuke 'authoring system' for arrogance and presumption.

Here's more bilge: "Curated by John Smith...the show will run...." The dunce who wrote 'curated' sinned twice: 1) Who 'curated' – a curate or a curator? 2) A 'curator' is the custodian of a museum; a 'curate', in the absence of a senior priest, is the custodian of a parish. Would anyone sane write 'custodioned by'? The writer meant the show was 'selected and organized by...' and should have said so.

Groups of humans 'migrate' from an old abode to a new one and intend to reside at the new. Birds and animals migrate periodically to feed or breed. Only pompous writers – searching for long, impressive words – migrate to a new computer environment. Do I migrate to the sanitation environment to upchuck?

Although 'read a good article' is clear and short, some oh-so-sensitive and with-it writers in the avant garde instead say, "The article was a good read". I now quote from an article on proofreading in the March, '89 issue of *Publish!* "...editing changes from the article's first read are keyed in and the manuscript is formatted...[and]...the formatted article is then given a second read..."

Given a second read indeed! I translate the 23 words of this precious twaddle into 16 of English: "On first reading, editors correct errors in the manuscript...after it's formatted, the editors proofread it again." If the sound of 'a good read' pleases you, dine at a good restaurant and tell the chef, "That was a good eat!"

Confusion by hyphen

Two generations of boobtubers don't understand how to use hyphens within a sentence (I'm not discussing hyphens at end-of-line here).

Compound modifiers of a noun usually demand a hyphen. Here's a first-rate example from a professional whodunit author, Elmore Leonard (in *Freaky Deaky*):

> "...a guy by the name of Booker, a twenty-five-year-old super-dude twice-convicted felon, was in his Jacuzzi..."

May I in future read that owners 're-cover' Amiga after disassembly; and not that they 'recover' it (from a thief?). I hope that no more 'used computer salesmen' are badly used, but that 'used-computer salesmen' sell used machines. I'd like to know if a 'long dead program' is a huge one – or long-dead? Last, please don't tell me about 'two year old drives' when you must mean either 'two-year-old drives or 'two year-old drives'.

Amputation by headline

Because headline writers never have enough space, *InfoWorld* substituted 'ship date' for the 'shipping date' of programs. It also shortened 'spelling checker' to 'spell checker'. Some bulkhead of every ship I've sailed on bears a bronze plate noting the date and place the ship was built. When told a 'ship date' has slipped, I see a bronze plaque clatter from a bulkhead. As to 'spell checkers': spells are magic invoked by witches and warlocks. What computer program ever laboured under a spell?

'Spell' is also an imperative verb that commands you to spell a word. When the title of a software box says 'Spell Checker', I refuse. Why in hell should I spell 'checker' when the word is already spelled? Because the noun 'spelling' defines what letters form words, only a 'spelling checker' finds spelling errors. Both 'ship date' and 'spell checker' are the bloody scalps of English massacred by illiterate barbarians.

Pomposity for its own sake

Writers use some words as appropriate as a dinner jacket at an urban riot. 'Environment' is an example: it means the totality of every influence and object that surrounds and impinges on its subject. Any environment is enormous. What do you include in a 'computer environment' – the computer, all your background and foreground programs, your desk and all gear on it, your chair and all room furnishings, electricity, all your reference books, the growl of distant traffic, the glug-glug of another JOLT Cola going down and the chatter of the TV, spouse and children?

If not, you don't mean 'environment', you mean 'setup': the way in which any apparatus is arranged and controlled. The word is short, clear, and simple. 'Environment' is constantly misused to impress readers. Trash the puckey below:

Instead of	say
in the home environment,	at home
in the office environment,	at the office
in the business environment,	in business
in the AmigaDOS environment,	when using AmigaDOS
in the CLI environment,	in CLI

Redundancies

Redundancies breed like horseflies: try 'append together'. Who in this world ever 'appended untogether'? In every line below, the first word (the verb) includes the meaning of the redundant second:

join together	assemble together
gather together	meet together
concatenate together	discuss together
shorten down	insert into
marry together (?!)	separate out

In the redundancies below, the writer obviously doesn't know what the first word or phrase means!

flurry of activity (flurry means activity or commotion)
current methods in use (methods not used aren't current)
a neophyte without experience (!!)
time period (the word "time" is defined as a period)

The redundancies below are pathetic attempts to append to a sentence the colour of metaphor. They're as useless as claims for virtues in a dead scoundrel – too little, too late and false. Nuke them; employ only the noun.

Instead of	use
in the business arena,	in business
in the educational field,	in education
in the programming area,	in programming
in the games scene,	in games
in the graphics stage,	in graphics
in the software front,	in software
in the communications sector,	in communications

Next, pure puckey. Who amongst you has ever seen a basic that was not fundamental, a foundation not essential – or learned anything without an experience?

basic fundamental	basic essential
fundamental foundation	basic foundation
essential foundation	learning experience

The new jargon

This generation can't seem to invent good new words. We desperately need a new, short word to mean 'write on a computer'. Instead, we have a horror: 'word processing'. Because we don't invent crisp, new words, we combine old words into a pompous computer jargon that has become a blight.

The infectious phase of this jargon is BLOPISH, an acronym for Bloated, Pompous, Inflated English. The second and ultimate form is High Abyssinian Greek (HAG). BLOPISH you can decrypt if given time; HAG is forever incomprehensible.

I illustrate the difference. The editor of *Dr. Dobbs* parodied BLOPISH in his July, '88 issue: a book is now 'multitasking, windowing pageware'. In contrast, you can read HAG in *ComputerWorld* every week. Try this example: "Despite the allocation of programmers to implement reforms prompted by the Tax Reform Act, a voice-prompted data-collection project was implemented without delay". Despite the programmers, note.

The blight is worst in computer manuals, handbooks and journals that sin by pride and ostentation. Larry Constantine, in the April '89 *BYTE*, notes that "the computer field likes big words". 'Instantiation', he says, is preferable to 'instance', and 'methodology' clearly superior to 'method'. Constantine remarks that 'methodology' means the study of methods, and that 'software methodology' is gibberish.

He's right. All my reference books say that methodology is the 'science of method or arrangement.' You can study software methods using the science of methodology. Alas, in a brochure from Boston University, I'm told of an upcoming seminar on the "most talked-about programming methodology", object-oriented programming. That should wither whatever League Ivy still clings to the walls of Boston U.

BYTE and other computer journals inflate their egos by always substituting 'paradigm' for the simple synonym 'model', and infinitely prefer to misuse 'parameter' (an oft-raped word) to mean 'variable'. A parameter in math is a constant quantity to which you assign an arbitrary meaning for a specific case.

Here's a simple example of parameters and variables: If you draw a set of concentric circles, the center of the set is a parameter for the set; the radius of each circle is a parameter for that specific circle. The colour of each circle and the spacing between circles are variables. Neither colour nor spacing, however varied, destroys the set of concentric circles. But should you specify a set of red concentric circles, red becomes a parameter of the set.

Although proper use isn't clear, I prefer to 'pass arguments' to a program. If you say you 'pass parameters', you're pompous and often wrong. (The 'argv' and 'argc' of C tend to confirm this view.)

Above all, don't write 'parametric variations', as many programmers do. 'Parameter' emerges from the Greek *metron*, measure. Parametric arises from *metra*, and means 'near the uterus'.

BLOPISH seems to be admired and emulated, even by PR hacks who should know better. The lead paragraph of a press release from Shannon Associates says its new program, METAMORPHOSIS, is "a generic utility program which facilitates the transformation of any syntactically reducible character-oriented file to any other form while preserving the synonymity thereof".

Are academics any better at writing about computers? In the gas below, two Ph.D.'s of English discuss word processing: "Authors may choose to construct idea banks of character sketches and landscape descriptions, allowing them to catalogue more spontaneous impressions and other short pieces whose freshness will also reflect an on-the-spot fidelity to experience. The American transcendentalists David Thoreau and Ralph Waldo Emerson, for example, made the practice habitual". This is a BLOPISH paraphrase of Emerson's advice: "To preserve striking events, keep a good journal.

To read a first-rate piece of BLOPISH, screw up your courage and scan "The CASE Experience" in the April, '89 issue of BYTE. At times, the author soars to the heights of HAG: "These toolkits provide automated assistance for the design of databases and files by performing functions such as logical data modelling, conversion to third normal form, and generation of database schemas and program-code file descriptions".

Decoding BLOPISH

I now reveal a priceless secret: how to convert BLOPISH to plain English. I stumbled onto the secret because my printing bills always exceeded my budget. Faced with bankruptcy, you learn to pare words to the bone of meaning.

In step one, remove all poobah terms – 'implement', 'effect', 'employment', 'undertake' – and trash that orotund puckey-piler, 'consideration'. In step two, recognize that BLOPISH always smothers its verbs to death. How do you smother a verb? Append to it a deadly pillow of '-ize', '-ibility', '-ing', '-ible', or '-tion'. Murder verbs and you slaughter meaning.

I now translate the last sentence and its verbs into BLOPISH: "By the means of murdering verbs through the employment of their weaker forms, you may involuntarily effect the slaughtering of your intended meaning". Six words expand to 23 when I murder verbs in BLOPISH.

To resurrect BLOPISH text, revive the verbs (or substitute better ones) and burn the padding. I demonstrate how in the examples below, all taken from computer magazines or manuals. The examples are short to shield readers from an intolerable level of anal gas.

- "Both software *development* and software *maintenance will be done* without programmers." Translation: "We'll develop and maintain software without programmers."
- "Those who *plan to undertake* some *word processing*..." Translation: "Those who plan to write..."
- "The new system we have *implemented* will not have any trouble *handling* the expected deluge of tax return *processing*." Translation: "Our new system will process the deluge of tax returns easily."
- "The program *is to be commended* as a *learning experience* in the *language arts*." Translation: "The program teaches English commendably." 'Learning experience' and 'language arts' are pompous pufferies affected by teachers.
- "I *effected* an *improvement* in the program by *rendering* a *transformation* in the graphics." Translation: "I improved the program with better graphics.
- "We *effected* proper *processing* and *handling* of new orders in a *timely manner*." Translation: "We handled new orders quickly."
- "...to *effect* a warm boot *restart* with Amiga..." Translation: "To warm boot Amiga..."
- "What the tool does is the fundamental *selection consideration*." Translation: "Select the tool you need."
- "An important *consideration* in *implementing* CASE is a plan for *educating* users in CASE *tool and software development methodology*." Translation: "To succeed with CASE, train its users in the complex tools and methods it employs."

Those who resurrect an article from its BLOPISH state can chop a text to half its original length. Any magazine can then hold twice as many articles – or can cut its text pages by half. BLOPISH not only baffles, it costs a lot of money!

I hope that this article is taken to heart by the readers of *Transactor for the Amiga*, for they're the people who write much of what is printed about Amiga. *TransAmi* is by far the best of all Amigan journals in the skill of its authors and editors. But, alas, a number of examples in this article I found in its pages. I say nothing about other Amigan journals because I'd be sued for slander and libel.

Enough of tragedy; on to the comedy!

I read myself to sleep with trashy whodunits; in one, the villain's mistress pours a "vile of acid" into a timing device for a bomb. 'Vile' as a homonym for 'vial' inspired a new collective noun, equal in its power to evoke an image to 'prides of lions' or 'gaggles of geese':

Viles of illiteracy

From *The Rockets' Red Glare*, a book published in New York: "...the heavy rain poured off his pancho". Friend Pancho needs a poncho.

"She took packets of sulfur from her haversack...and dumped the yellow powder into the wound". The author means 'sulfa', a powdered drug carried by U.S. soldiers in World War II to prevent infection in wounds.

From the evening TV news: "He evoked his constitutional rights". Then, during the sports news: "He's a suede and sophisticated competitor". I hope he's suave, too. The next week, I heard that the governor received a "standing ovulation". (From a women's club, no doubt.) I sympathize with the Marine who was reported arrested as a "deserter from the Army". Any Marine in the Army has a constitutional right to desert.

From a user group newsletter: "I found the parts in a shopping maul". From a second newsletter "...the pieces are diced into dyes". I'd expect the pieces to be diced into dies, but some people dote on colour.

From the docs on a shareware program named *DirMaster*: "If you are unwilling to pay $25 for DirMaster, but still want to use it with a clear conscious, you can get group rates..."

From the MicroIllusions program, *Romantic Encounters at the Dome*: "...you move to her side and lay beside her..." Lay? In context, to 'lay' means to have sexual congress with (whom?). The writer means 'lie'.

From a teacher: "After ten years of teaching, I've lost patients with the lack of discipline in the classroom." Does he hold a second job as a surgeon and operate in the classroom?

From an April '89 *InfoWorld*: "...heavy traffic bogs a [highway] system down. You can be parked on an immobile highway, even if..." Has anyone ever seen a mobile highway? Aircraft carriers I disqualify.

From a May *InfoWorld*: "...a consultant is currently automating a cosmetics firm." Androids, arise! The same article says that a "hard disk scrambled"– as does a quarterback? Or in the manner of eggs?

From the *Amiga Sentry*, March 1989, three full viles: 1) "Up to four bit plains (16 colours) can..." 2) "Standing in your way are the Queen's evil guards and viscous Death Dogs". Can a viscous Death Dog run as fast as cold maple syrup? 3) "Cynthia tells Crash just enough to...peak his interest." Piqued interest I can understand. Does 'peak' imply that ole Crash got a visible hard on?

From *ComputerWorld* on April 24: "...the model 400E is a tough machine to get ahold of." As any redneck can tell you, the proper form is: "git aholt of."

From a user group newsletter: "The door prizes for attending our last meeting were bags of home-made whorehound candy." The homonym is horehound (minty and good). I describe this vile by the homonym it deserves: a whorer!

Curmudgeonry suspended

I've shown that much of the computer press is pompous, verbose and illiterate by quotation. TV, newspapers and magazines are almost as bad. Had I ten more pages and time to cover plurals, apostrophes and possessives of English, the gutters would run with blood.

ANNOUNCING!!

THE SERIAL SOLUTION

a dual-port internal serial card for the Amiga 2000

- the *only* serial board with an Amiga-compatible 25-pin serial port
- the 25-pin port supplies +-12 volts for specialized Amiga peripherals
- the second port is a 9-pin AT-compatible serial port
- supports a variety of devices including:
  printers,
  modems,
  MIDI interfaces,
  sound samplers,
  drawing pads,
  VCR controllers,
  and other serial devices
- works with most existing software (requires small patch)
- compatible with SER: and "serial.device"
- Full 120-day manufacture's limited warranty

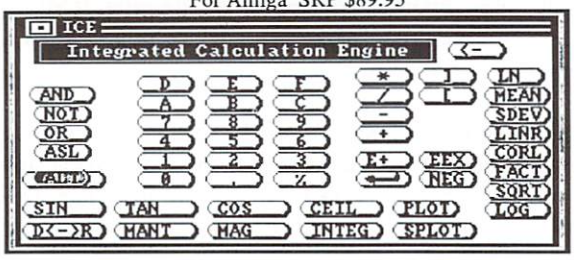
CHECKPOINT
TECHNOLOGIES
P.O. Box 2035 • Manassas, Virginia 22110 • 703-330-5353

Distributor Inquiries Welcome

Amiga is a trademark of Commodore-Amiga, Inc. AT is a trademark of IBM.
THE SERIAL SOLUTION is a trademark of Checkpoint Technologies.

I.C.E.

Integrated Calculation Engine
For Amiga SRP $89.95



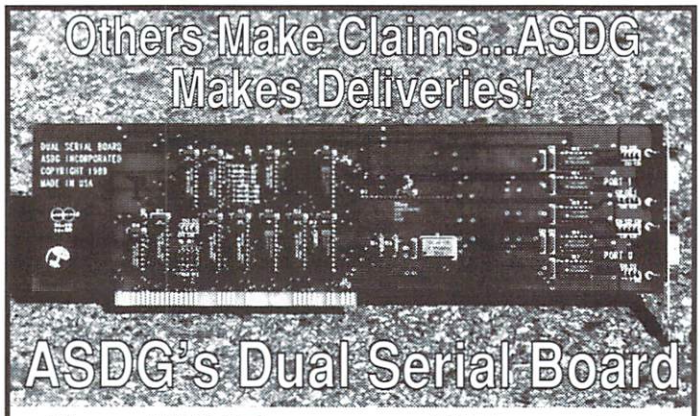
Who needs it?

Students Programmers Scientists Teachers Financiers Statisticians Businessmen Accountants Researchers Technicians Realtors Leasors Rental agents Contractors Engineers Mathematicians Numbercrunchers Everywhere!

Why?

I.C.E. makes number work easy! I.C.E. is a programmable calculator construction set. Create calculators designed to do the jobs you need done. Using the mouse, select the buttons (functions) you want for your customized calculators. Add the graphic plot &print functions, choose the layout and number system you find comfortable, bind macro programs to a single key for truly powerful versatility. Then use your I.C.E. creations with other programs like spreadsheets and databases. I.C.E. is not copy protected and multitasks beautifully! Now you can have programmable calculators with the features you choose. I.C.E. calculators include an integral program editor which allows you to create complex formulae from over fifty available functions.

KING PUBLISHING
921-32nd Street Sioux City IA 51104 Phone 712-252-4604



Use *ANY* Program Written For "serial.device" On *ANY* Expansion Port.

● Two high speed RS-232C serial ports for the Amiga 2000. ● IBM PC AT style (9 pin male) connector for each port. ● Full 7 wire implementation plus Ring Indicator and DCD. ● Send and Receive 110 to 76800 baud (and higher). ● Complete Amiga system software (both Exec *and* DOS). ● Using our exclusive Serial DisPatcher, programs can use any port *without* modification. ● Public domain and shareware software on the distribution diskette.

ASK YOUR DEALER!

ASDG
925 Stewart Street
Madison, Wi. 53713
(608) 273 - 6585

Amiga is a trademark of Commodore-Amiga, Inc. IBM PC AT is a trademark of IBM Corp.

Getting A Handle On File Systems

Introducing the Software Distillery handler code

by John Toebes
Copyright © 1989 John A. Toebes, VIII

John Toebes is Manager of the Motorola Compiler Department at SAS Institute, Inc., responsible for the recent release of the Lattice C compiler. He is also coordinator for the Software Distillery. He can be reached on Bix (JTOEBES) and Plink (JTOEBES) as well as Usenet (mcnc!rti!sas!toebes).

Having talked last issue about file systems and handlers in general, we now take up the topic of mapping the file system to the physical media. As an example, we will construct a file system that is compatible with the Old File System currently used for Amiga floppy disks.

With our handler, there are only a small number of source files that understand how to manipulate the file system layout. Working from the bottom layer up, we have:

Physical level

Disk.c: Routines for reading from and writing to the physical media. These routines only understand the concept of fixed-sized blocks, and do not know anything about the data contained within the blocks. There is no buffering done at this level.

Block buffer level

BitMap.c: Routines to manage the bitmap for allocation of blocks.

DataBuff.c: Virtual buffering scheme for caching of the disk blocks in memory. Only data blocks are managed here; those blocks assigned to store the bitmap are managed by BitMap.c

Disk structure level

Mount.c: The mount and unmount routines, which verify a diskette and make it available to AmigaDOS.

Dos1_2.c: By far the largest of the source files. This contains all the access routines for managing the file and directory structure within the blocks.

This layering allows us to separate out the various functions for manipulating the media so that we can readily replace or refine those operations. For example, with the physical-level routines, we could implement a handler that talks directly to a specific piece of hardware (such as a CD-ROM or an alien disk drive) without having to change the layers above.

Before we examine these routines, there are two important pieces of information to be aware of: All routines in the handler take a parameter called *global* as their first argument. This is a pointer to a global data area shared by all routines, and is used to allow the handler to be fully reentrant. The memory for this area is allocated in *main.c* and is passed on down.

Another common parameter used throughout the handler is a *key*. This is a 32-bit long value that indicates a block number. A key of 0 is a special case indicating a non-existent key or, when returned, a failure to locate an object. In the case of the Dos1_2 handler, this key is actually a physical block number, while in the MS-DOS handler it is actually a combination of a block number and a physical offset within that block. This representation is necessary because more than one directory entry can occur within a physical block on an MS-DOS diskette.

The physical level

The physical level performs two basic operations. The simplest is catching disk-change interrupts so that the handler can become aware of the presence of a new volume. The big operation, however, is the actual transfer of data from the internal buffers to the physical media. The routines responsible for this are:

ReadPhy(global, buf, key)
WritePhy(global, buf, key)

ReadPhy reads data from the disk into the specified buffer; *WritePhy* writes the contents of the buffer to the disk. The *key* parameter indicates the block number on the disk to be manipulated. Above this layer, all routines assume that the media is made up of a number of fixed-sized blocks. It is the responsibility of these routines to map this number onto the appropriate track, sector and head for the media.

These routines are assumed to be synchronous, returning only when the operation has been completed. This is important in the case of the *WritePhy* routine, as it is used to ensure that data are flushed to the disk.

Block buffer level

Sitting on top of the physical-level routines are two sets of routines for managing the buffers corresponding to the blocks on the disk. This is separated into two source files to distinguish the usage of the media.

The BitMap routines are responsible for managing which blocks on the disk are currently in use. All higher level routines that wish to write to the media must call:

AllocateBlock(global, key, type)

to get space on the media. The *key* parameter is a locality indicator that *AllocateBlock* takes into account when trying to create a new block so that it can be as close as possible. The *type* parameter indicates what the block will be used for. A naive implementation of this routine could readily ignore both these parameters and just get the next free block. A corresponding routine is used to indicate that a given block is no longer in use:

FreeBlock(global, key)

Also contained within *BitMap.c* are routines to initialize the bitmap and to flush it to the disk. These are called whenever a disk has been inserted or a request has been made to flush all buffers.

DataBuff.c complements the bitmap routines by providing a caching scheme to manage the blocks in memory:

GetBlock()
ModBlock()

These routines implement a virtual buffering scheme in which only one block is allowed to be accessed at a time. When a higher-level routine wants a specific block from the disk for reading, it calls *GetBlock*. The pointer returned is only valid until the next call to *GetBlock*. Because block pointers are not kept around, the buffers may be flushed, moved or even reallocated at any time without having to inform any other routine.

If for any reason the block is not available (as in the case of a read error), these routines return NULL and set the global error code. It is the responsibility of all routines to check the result code and fail appropriately. The scheme used in *DataBuff.c* is a simple list of all available blocks along with a status for each block. If a block is requested that is not currently in memory, it finds an empty buffer (flushing one to disk if necessary), and then calls *ReadPhy* to get the data from the media. *ModBlock* does the same process, with the additional step of setting a dirty bit for the buffer to ensure it is flushed out later. (People familiar with Forth block management will recognize this method.)

At this level, there is still no concept of layout of data on the disk - only a simple management of blocks on the physical media.

Disk structure level

The real meat of the handler occurs at the disk structure level, in a single source file: *Dos1_2.c*. Although a number of subroutines in this file are quite small, they are separated out to hide all manipulation of the physical layout. By changing these routines alone, it is possible to greatly affect the physical layout. In fact, we do just this to create the MS-DOS handler.

We can divide these routines into several categories:

Object information

prot = GetProtect(global, key)
Return the protection bits associated with a file entry.

*rc = SetProtect(global, key, long /*Protection*/)*
Set the protection bits for a file.

type = GetType(global, key)
Determine the type of a file entry.

rc = SetCommentStr(global, key, comment)
Set the comment string for a file.

*rc = GetInfo(global, key, struct FileInfoBlock *)*
Fills in an AmigaDOS FileInfoBlock structure with information about the file indicated by the key.

As a demonstration of the simplicity of implementing these routines, the source for *GetProtect*, *SetProtect*, *SetCommentStr*, and *GetInfo* can be found in Listing 1 at the end of this article. It is through the use of *GetBlock* to perform virtual management that these routines are so easy to write.

Entry manipulation

These are by far the most important routines. They provide the only method by which the higher-level routines can convert a name into a key corresponding to the given object.

These routines (shown in Listing 2) understand how to parse file names and to traverse directories. *CreateEntry* has the additional responsibility of validating the protection on any existing entries before overwriting them.

*key = LocateEntry(global, key, char * /* Name */)*
Locates a file within the directory structure of the disk. The input key is a directory from which the search is to start.

*key=CreateEntry(global,key,char *name,int mode,int type)*
Works like LocateEntry except that a new object is to be created. *mode* indicates whether a new file is to be created or an old file may be overwritten. *type* allows creation of directory or file objects.

As you can see by Listing 2, *LocateEntry* and *CreateEntry* are actually quite simple, relying upon the lower-level routines *FindDir* and *FindEntry* to perform the directory traversal. *FindDir* parses the name out, locating the target directory and leaving just the name portion. *FindEntry* picks up where it left off and actually locates the requested object. This separation lets us use the same directory traversal code both when locating and when creating an entry.

Object I/O

The *PUTDATA* and *GETDATA* routines were intended to be implemented as macros that hide the access to the data in the buffers. As it turns out, it is much more convenient to implement them as routines that do the moving. They provide the only means for the I/O level routines to get data in and out of the physical buffers.

*PUTDATA(char *, long, char *, long)*
Stuff file data into a data block.

GETDATA(char *, long, char *, long)
Extract file data from a data block.

The next three routines use the concept of an 'Extended File Handle' (EFH) to indicate the file to be manipulated. This EFH is used as a place to store some information about the current location and extension block in order to improve performance by not recalculating this information each time.

size = GetFileSize(global, EFH)
Determine the current length of a file. This routine is primarily used by the I/O buffer routines to determine the location of the EOF mark in order to limit the number of bytes of I/O.

rc = UpdateFile(global, EFH, long, long)
When a given file has been modified, this routine is called to set the current EOF marker and the date/time stamp.

key = BlockKey(global, EFH, long, int)
This routine is the heart of the file access routines. Given a desired block number (relative to the start of the file), this routine returns the key associated with that block. If the block is past the end of the file and the operation is intended to be a write, this routine will automatically extend the file by a single block. This is the only routine that knows how to manipulate the side sectors that index the file.

Directory management

*key = NextKey(global, rootkey, struct FileInfoBlock *, key)*
Advances to the next directory entry. This is used to implement the ACTION_EXAMINE_NEXT packet in order to hide the hashing from the higher level routine.

The *rootkey* parameter indicates the directory to be searched, and *key* indicates the last entry that was examined. It is seeded with **key = 0** for the first call.

key = ParentKey(global, key)
Locates the Parent object for a given key. If no parent exists, the root of the volume is returned.

*rc = DeleteEntry(global, key, char *)*
Removes a directory entry, returning any associated blocks to the bitmap. This routine must examine protection bits to verify that the operation is allowed as well as ensure that directories are empty before permitting the operation.

*rc = RenameEntry(global, oldkey, key, char *oldname, char *newname)*
This is one of the most complicated routines. It implements most of the work for the ACTION_RENAME packet by renaming and potentially moving a directory entry. It is responsible for checking for invalid operations to prevent directory loops or orphaned directories.

In order to perform the rename operation, we first check for the frequent case of both original location and new location being in the same directory. Also, if the object is a file and not a directory, we can skip the extra checking. If not, we walk the parent chain of first the source and then the target to ensure that neither is an ancestor of the other. Only when we are certain that the operation is safe do we do it.

Other

*rc = RenameDisk(global, char *)*
Changes the name on the disk to the given name.

In addition to *Dos1_2.c*, there is a small amount of knowledge about the format of the physical media contained within *Mount.c*. The routine *mount* is called whenever a new volume is encountered to process the volume and to initialize the environment for the remaining routines. Although not implemented in our handler, this routine is responsible for kicking off a validator process if the media is found to be in an invalidated state. If, for some reason, the *mount* routine determines the media to be unusable, it will set a flag indicating so in global so that all lower-level routines will return failures when accessing it.

Putting it together

Given these routines and those provided last month, we now have the makings of a real file system. Let us take a look at how a simple operation such as:

```
echo >myfs:dir/name.x "file fodder"
```

interacts with our file system. This AmigaDos command will eventually be translated into three operations that are sent to us:

```
   ACTION             ARG1        ARG2          ARG3
   ----------------- ----------- -----------  -----------------
1) ACTION_FINDOUTPUT Filehandle  0             'myfs:dir/name.x'
2) ACTION_WRITE      FH->fh_Arg1 'file fodder' 11
3) ACTION_END        FH->fh_Arg1
```

Let us follow the logic in handling these packets:

ACTION_FINDOUTPUT is routed to *ActFindout* in *IO.C*.
- *ActFindout* calls *initbuf* for a SHARED_LOCK on a NEWFILE.
  - *initbuf* allocates and initializes an extended file handle.
  - *initbuf* calls *CreateEntry* for an OLDFILE of type ST_FILE.
    - *CreateEntry* calls *FindDir* to locate the entry.
      - *FindDir* parses out the **myfs:**, sees the **:** and steps to the root of the device.
      - *FindDir* parses out the **dir/** and locates the directory **'dir'** relative to the root directory
      - *FindDir* returns **'name.x'** and a key indicating the directory.
    - *CreateEntry* calls *FindEntry* to locate **'name.x'**.
      - *FindEntry* scans the directory but does not locate **'name.x'**.
    - *CreateEntry* calls *MakeEntry* to create the file.
      - *MakeEntry* validates that the location to put the file in is really a directory.
      - *MakeEntry* calls *AllocateBlock* to get a place to store the entry.
      - *MakeEntry* calls *LinkDir* to insert the file in the appropriate hash chain.
      - *MakeEntry* calls *ModBlock* to find the buffer in memory.
      - *MakeEntry* fills in the newly-allocated block and returns the key.
    - *CreateEntry* returns the key.
  - *initbuf* creates a lock for the file based on the returned key.
  - *initbuf* completes initialization of the EFH and returns the key.
- *ActFindout* returns the completion status.

ACTION_WRITE is routed to *ActWrite* in *IO.C*.
- *ActWrite* preserves the current file postion information and validates the file handle.
- *ActWrite* calls *BlockKey* for the first block in the file for write.
  - *BlockKey* notices that the request is one past end of file and calls *AppendDataChain*.
    - *AppendDataChain* calls *AllocateBlock* for a new block.
    - *AppendDataChain* stores the block pointer in the file chain array.
    - *AppendDataChain* calls *ModBlock* to find the buffer and initializes the data block to be empty.
  - *BlockKey* returns the newly allocated key.
- *ActWrite* calls *PUTDATA* to put 11 bytes into the start of the block.
- *ActWrite* calls *UpdateFile* to set the end of file mark and data stamp.
- *ActWrite* updates the current position information in the EFH.
- *ActWrite* returns the completion status.

ACTION_END is routed to *ActEnd* in *IO.C*.
- *ActEnd* validates the file handle.
- *ActEnd* calls *termbuf* to flush and close the file.
  - *termbuf* unlocks the lock associated with the file handle.
  - *termbuf* frees the extended file handle.
- *ActEnd* returns the completion status.

Later on, a timer event will occur and the buffer routines will be called to flush the buffers to disk. At this time all buffers will be marked as flushed to prevent them from being written out again.

Implementing other file systems

We have now seen how an AmigaDos-compatible handler works. Because of the relatively few dependencies on the actual media, we can replace the routines in *DOS1_2.c* and come up with our own format. One could readily implement an MS-DOS type format or even experiment with the existing format to find better methods of storing directory entries, or even ways of squeezing more data on the disk. Other things to try are buffering methods and caching schemes to improve performance in that area. The possibilities are endless - what can you do?

Acknowledgements

The Software Distillery file system is a collective effort of the Distillery, with major contributions by John and Doug, John Mainwaring, Jim Cooper, Dave Baker, Bruce Drake and Gordon Keener. Special thanks are due to Steve Beats and Andy Finkel of Commodore Business Machines for putting up with our nitnoid questions.

Correction

The parameters we gave for ACTION_SET_DATE in Volume 2, Issue 2 were incorrect. The correct specification for the packet is:

```
ACTION_SET_DATE  34  <sendpkt only>
ARG1:    - Unused
ARG2: LOCK - Lock to which name is relative
ARG3: BSTR - Name  of object
ARG4: BPTR - DateStamp

RES1: BOOL - Success/failure
RES2: CODE - Failure code if RES1 = DOSFALSE
```

As this packet has always been a source of some confusion, we hope that you can understand the mix-up.

Listing 1: Object information routines

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Routine: GetProtect                                                    */
/* Purpose: Return the protection bits associated with a file entry       */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
int GetProtect(global,key)
GLOBAL global;
KEY key;
{
struct DirBlock *block;

   block = (struct DirBlock *)GetBlock(global,key);
   if (block == NULL)
      return(FIBF_WRITE|FIBF_READ|FIBF_DELETE|FIBF_EXECUTE);

   return(block->db_Protect);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Routine: SetProtect                                                    */
/* Purpose: Set the protection bits for a file                            */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
int SetProtect(global, key, prot)
GLOBAL global;
KEY key;
long prot;
{
struct DirBlock *block;

   if (key == 0 ||
       (block = (struct DirBlock *)ModBlock(global,key)) == NULL)
      return(DOS_FALSE);

   block->db_Protect = prot;
   return(DOS_TRUE);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Routine: SetCommentStr                                                 */
/* Purpose: Set the comment string for a file                             */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
int SetCommentStr(global, key, comment)
GLOBAL global;
KEY key;
char *comment;
{
   struct DirBlock *block;

   if (key == 0 ||
       (block = (struct DirBlock *)ModBlock(global,key)) == NULL)
      return(DOS_FALSE);

   memcpy((char *)block->db_Comment, comment, (*comment)+1 );
   return(DOS_TRUE);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Routine: GetInfo                                                       */
/* Purpose: Copy the file information for a dos info block                */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
int GetInfo(global, key, info)
GLOBAL global;
KEY key;
register struct FileInfoBlock *info;
{
   register struct DirBlock *block;
   char *p;

   /* get the block associated with the key */
   if ((key == 0) ||
       ((block = (struct DirBlock *)GetBlock(global,key)) == NULL))
      return(DOS_FALSE);

   /* It would be prudent here to check the type of block and see that it */
   /* Is a true file entry block                                          */

   /* Now copy over the information they want */
   info->fib_DiskKey        = key;
   info->fib_EntryType      =
   info->fib_DirEntryType   = block->db_SecondaryType;

   /* Set the entry type to 2 to make dir happy */
   if (info->fib_EntryType == 1)
      info->fib_EntryType      =
      info->fib_DirEntryType   = 2;

   p = (char *)block->db_Name;

   memcpy(info->fib_FileName, p, (*p)+1);
   info->fib_Protection     = block->db_Protect;
   info->fib_Size           = block->db_Size;
   info->fib_NumBlocks      = (block->db_Size + (BLOCKSIZE-1))/BLOCKSIZE;
   info->fib_Date.ds_Days   = block->db_Days;
   info->fib_Date.ds_Minute = block->db_Minutes;
   info->fib_Date.ds_Tick   = block->db_Ticks;

   p = (char *)block->db_Comment;
   memcpy(info->fib_Comment,  p, (*p)+1 );

   return(DOS_TRUE);
}
```

Listing 2: Entry manipulation routines

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Routine: FindDir                                                       */
/* Purpose: Find a Directory associated with a file and parse out the name */
/*          of the file from the file string                              */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
KEY FindDir(global, key, namep, lenp)
GLOBAL global;
KEY key;
char **namep;
int *lenp;
{
int len;
char *p, *t;
register struct DirBlock *block;
char namebuf[32];
int hashval;

   len = *lenp;
   p = *namep + len - 1;

   /* Skip to the last colon or / in the name */
   while ((*p != ':') && (*p != '/') && len) { p--; len--; }
   *lenp -= len;

   t = p;
   if (len) t++;
   /* Are they attempting to access '.' or '..' ? */
   if (*t == '.' &&
       ((*lenp == 1) ||
        ((*lenp == 2) && (*(t+1) == '.'))))
      {
      /* Yes, so treat it as part of the directory */
      len += *lenp;
      *lenp = 0;
      }

   BUGLSTR("Locate: Path is :", *namep, len);

   /* Now run through the string searching the directory */
   while(len && key)
      {
      /* first go to the root of the directory they asked for */
      block = (struct DirBlock *)GetBlock(global,key);
```

```
    if (block == NULL)
       {
       BUG(("FindDir: Error accessing block %ld\n", key));
       return(0);
       }

    if ((block->db_SecondaryType != ST_ROOT) &&
        (block->db_SecondaryType != ST_USERDIR))
       {
       /* they want to traverse a file as a directory - give them hell */
       global->pkt->dp_Res2 = ERROR_OBJECT_WRONG_TYPE;
       return(0);
       }

    /* break down the next subcomponent of the name */
    hashval = parse(namep, &len, namebuf);

    BUGBSTR("Parsed Name :", namebuf);
    BUG(("Hashval = %ld\n", hashval));

    /* Did they specify the root directory ? */
    if ((namebuf[0] == 1) && (namebuf[1] == ':'))
       {
       key = global->Root;
       BUG(("Moving to root: %ld\n", key));
       }

    else if ((namebuf[0] == 0) ||
            ((namebuf[0] == 2) && (namebuf[1]=='.') && (namebuf[2] == '.')))
       {
       key = block->db_Parent;
       BUG(("Moving to paremt: %ld\n", key));
       }

    else if ((namebuf[0] != 1) || (namebuf[1] != '.'))
       {
       /* Search the current block for the given name */
       for( key = block->db_HashTable[hashval]; key != 0;
            key = block->db_HashChain)
          {
          block = (struct DirBlock *)GetBlock(global,key);
          if (block == NULL)
             {
             BUG(("FindDir: Can't get block %ld\n", key));
             return(0);
             }

          BUGBSTR("Block name:", block->db_Name);
          if (IsSameName(block->db_Name, (char *)namebuf+1, namebuf[0]))
             break;
          }
       }
    }

  if (key == 0)
     global->pkt->dp_Res2 = ERROR_OBJECT_NOT_FOUND;

  BUG(("locate: key is %ld\n", key));
  return(key);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Routine: FindEntry                                                       */
/* Purpose: Locate an entry in a directory                                  */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
KEY FindEntry(global, key, name, len)
GLOBAL global;
KEY key;
char *name;
int len;
{
int hashval;
struct DirBlock *block;

  BUGLSTR("FindEntry: Name is ", name, len);

  if (len == 0)
     {
     BUG(("Length is zero, returning key=%ld\n", key));
     return(key);
     }

  hashval = hash(name, len);
  /* first go to the root of the directory they asked for */
  if ((block = (struct DirBlock *)GetBlock(global,key)) == NULL)
     return(0);

  BUG(("Starting search for hash=%ld from key=%ld\n", hashval, key));

  if ((block->db_SecondaryType != ST_ROOT) &&
      (block->db_SecondaryType != ST_USERDIR))
     {
     BUG(("Secondary type %ld bad for key %ld\n",
          block->db_SecondaryType, key));
     /* they want to traverse a file as a directory - give them hell */
     global->pkt->dp_Res2 = ERROR_OBJECT_WRONG_TYPE;
     return(0);
     }

  global->prevkey = 0;

  /* Search the current block for the given name */
  for( key = block->db_HashTable[hashval]; key != 0;
       key = block->db_HashChain)
     {
     if ((block = (struct DirBlock *)GetBlock(global,key)) == NULL)
        return(0);

     BUGBSTR("Block name:", block->db_Name);
     if (IsSameName(block->db_Name, name, len))
        break;

     global->prevkey = key;
     }

  if (!key)
     global->pkt->dp_Res2 = ERROR_OBJECT_NOT_FOUND;

  return(key);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Routine: MakeEntry                                                       */
/* Purpose: Create a new entry for a file                                   */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
KEY MakeEntry(global, key, name, len, type)
GLOBAL global;
KEY key;
char *name;
int len;
int type;
{
int hashval;
KEY newkey, nextkey;
struct DirBlock *block;
LONG *zeroptr;
zeroptr = 0;

  BUGLSTR("MakeEntry: name ", name, len);

  hashval = hash(name, len);

  /* first go to the root of the directory they asked for */
  block = (struct DirBlock *)GetBlock(global,key);

  /* When all else fails pop up to let it be handled */
  if (block == NULL) return(0);
```

```
    if ((block->db_SecondaryType != ST_ROOT) &&
        (block->db_SecondaryType != ST_USERDIR))
       {
       /* they want to traverse a file as a directory - give them hell */
       global->pkt->dp_Res2 = ERROR_OBJECT_WRONG_TYPE;
       return(0);
       }

    /* Get us a block to put the newly created item */
    if (!(newkey = AllocateBlock(global, key, type)))
       {
       BUG(("MakeEntry:Unable to allocate a block %ld\n", newkey));
       global->pkt->dp_Res2 = ERROR_DISK_FULL;
       return(0);
       }

    /* try to put block into right place in parent directory */
    if ((nextkey = LinkDir(global, key, newkey, hashval)) == -1)
       {
       FreeBlock(global, newkey);
       return(0); /* invalid key - admission of failure */
       }
    BUG(("MakeEntry: new entry %d points to successor %d\n", newkey, nextkey));

    /* Find the newly created block and fill it in */
    block = (struct DirBlock *)ModBlock(global, newkey);
    if (block == NULL)
       {
       /* Actually getting an error here is quite fatal because we already   */
       /* modified the directory entry.  We need to figure out a way to back */
       /* out of the creation.  The only situation that should cause this is */
       /* if the allocated block gets flushed out to disk and we are unable  */
       /* to read it in.  In this case it is quite likely that the media is  */
       /* not usable.  However we eventually may have to handle this case    */
       FreeBlock(global, newkey);
       return(0);
       }
    memset((char *)block, 0, sizeof(struct DirBlock));

    BUG(("MakeEntry: New entry %ld at %08lx\n", newkey, block));
    block->db_HashChain = nextkey;

    /* Fill in the remainder of the information */
    block->db_Type              = T_SHORT;
    block->db_OwnKey            = newkey;
    block->db_SecondaryType     = type;
    memcpy(((char *)block->db_Name)+1, name, len);
    *(char *)&block->db_Name[0] = len;
    block->db_Parent            = key;
    SetDateInfo(global, block);
    return(newkey);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Routine: LocateEntry                                                      */
/* Purpose: Find an file entry                                               */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
KEY LocateEntry(global, key, name)
GLOBAL global;
KEY key;
char *name;
{
int len;

len = *name++;

BUGLSTR("LocateEntry: Full Name is:", name, len);
if (!(key = FindDir(global, key, &name, &len)))
   return(0);

BUGLSTR("LocateEntry: name ", name, len);

return(FindEntry(global, key, name, len));
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Routine: CreateEntry                                                      */
/* Purpose: Create a new file entry                                          */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
KEY CreateEntry(global, key, name, mode, type)
GLOBAL global;
KEY key;
char *name;
int mode;
int type;
{
int len;
KEY newkey;
struct DirBlock *block;

len = *name++;
BUGLSTR("CreateEntry: Name ", name, len);

if (!(key = FindDir(global, key, &name, &len)))
   return(0);

BUGLSTR("CreateEntry: Parsed Name ", name, len);
if (newkey = FindEntry(global, key, name, len))
   {
   /* Entry already exists, may need to blow it away */
   if (mode == NEWFILE)
      {
      global->pkt->dp_Res2 = ERROR_OBJECT_EXISTS;
      return(0);
      }
#if 0
   /* Although logical course of action, the current filing system seems */
   /* to not exactly work this way.                                      */
   if (GetProtect(global, newkey) & FIBF_DELETE)
      {
      global->pkt->dp_Res2 = ERROR_DELETE_PROTECTED;
      return(0);
      }
#endif
   if (GetProtect(global, newkey) & FIBF_WRITE)
      {
      global->pkt->dp_Res2 = ERROR_WRITE_PROTECTED;
      return(0);
      }
   if ((GetType(global, newkey) == ST_USERDIR) || (type == ST_USERDIR))
      {
      global->pkt->dp_Res2 = ERROR_OBJECT_WRONG_TYPE;
      return(0);
      }
   FreeDataChain(global, newkey);

   /* Also we need to update the name information in the file          */
   /* This allows them to change the case of the name.  Note that by   */
   /* doing this in place we preserve all the comment information and  */
   /* attributes of the old entry.  Also since the entry doesn't move on */
   /* disk we can keep it in the same plae on the hash chain.          */
   block = (struct DirBlock *)ModBlock(global, newkey);
   if (block != NULL)
      {
      memcpy(((char *)block->db_Name)+1, name, len);
      *(char *)&block->db_Name[0] = len;

      /* Clear the archive bit so backup programs know it has been modified */
      block->db_Protect &= ~FIBF_ARCHIVE;
      }
   }
else
   {
   /* We have a brand new file here, so make us an entry for it */
   newkey = MakeEntry(global, key, name, len, type);
   BUG(("CreateEntry: newkey is %ld\n", newkey));
   }
return(newkey);
}
```

Effective Documentation

A standard approach to writing user manuals

by Rick Stevens
Copyright © 1989 Rick Stevens

Rick Stevens has been in the computer field for 17 years. He currently works for TMI/MacroTest in Irvine, California, where he holds the title of Excel Software Projects Manager. In his spare time he programs his two Amiga computers, manages his consulting firm and a recording studio, and plays guitar and drums. He can be reached at his CompuServe PPN of 75006,1355.

1. Introduction

Okay. You've just written a new program that will make the Amiga do backflips, roll over, speak, balance your check book, do your taxes, feed the kids, bring you your slippers, make a terrific Martini, and cook you a gourmet dinner. Now it's time to write the manual. You groan, grumble, and utter that famous quote, "Ah, they don't need a manual. It's so simple to use, a two year old can do it." Well, bucko, not too many two year olds can buy your program. It's going to be adults who buy it, and we adults are notoriously stupid. If you don't believe me, ask any teenager.

Remember that first impressions are important. Your company and its products are more often judged by a customer's impression of the quality of the products than by technical prowess, specifications, or price. The average customer is going to want the program demonstrated. If he's interested, he'll want to see the manual. If the manual is shoddy, goodbye sale! You will not be a happy camper.

Customers are going to spend a good deal of hard-earned money when they purchase your products. They want to feel that the company providing the product is a professional and reliable firm. One of the best ways to accomplish this is by adopting a documentation standard, and adhering to it as much as possible.

When these people need another program to perform some task, your products will get the first look, because they will know that they will be able to find anything they need to know in the manual.

To paraphrase an old cliché, 'Familiarity breeds consent!'

1.1 Organization of this document

This article covers a number of document style aspects you should consider when writing your user manuals. Some of the topics are:

page layout
margins and rulers
titling
section numbering
print enhancements
type fonts
graphic images
table of contents
glossaries
indexing
editorial style

Obviously, it doesn't cover everything, but it does give guidelines which can be extrapolated to deal with those instances not dealt with explicitly. Essentially, if you 'keep the flavor' set forth in this article, your documents will be fine.

1.2 Document contents

When writing a document, keep in mind your target audience. If you write for a sophisticated audience, the less knowledgable will be lost. If you write for neophytes, the professionals will get bored. Here are two basic targeting rules:

- If your audience is fairly knowledgable about the subject, remember that all you are doing is showing them how you are dealing with it.
- If they are somewhat less knowledgable, you might wish to include some tutorial information. In fact, you can have a tutorial chapter or section containing enough information for neophytes so that they can make use of the later sections. Furthermore, you may have a 'quick reference' section for really professional users.

The second approach is much better – that of a tutorial section that gives basic background on the subject and later sections that build on that foundation.

A point should be stressed here: Don't insult your reader's intelligence! No one is stupid, just less familiar with the subject matter than you are. Remember that, since you are writing the document, you are the expert.

You are going to be a teacher, and your job is to bring your reader up to your level. It may never happen, but it is a laudable goal.

Remember when you were struggling with some subject in school? Suddenly, either the textbook or teacher clarified some point, and, WHAM!, suddenly it all made sense. Remember that feeling? Well, that's what you are trying to do with your audience. When it happens, it's a great feeling!

This article uses less formal language than the documents you may write. Since it is targeted towards authors, it can be a bit more personal than you can be when targeting a user audience.

Now, let's move on to the meat and potatoes of this article.

2. Page layout

The layout of a page is very important. It establishes the visual appearance of the document, gives it unity, and determines its readability.

The factors that make up a page layout are the paper form factor, margins, number of columns, column width, gutter width, headers and footers, and page number placement. Essentially everything you notice when you take a cursory glance at a page make up the page layout.

2.1 Paper form factor

There are about as many paper form factors as there are trees. However, there are certain standards one expects in any given country. Find out what is common in your intended market, and use that form factor.

The normal paper size used in the United States and Canada is 8.5 inches wide and 11 inches long. Paper of this size is called by several different names such as 'letter', 'A-size' and 'binder'.

Smaller binders which fit into slip covers (such as those used by IBM-PC software and hardware reference manuals) use half-sheets of paper; their page size is 5.5 inches wide by 8.5 inches long. This is a popular format, but requires special binders and most copiers can't handle the paper size well.

2.2 Top and bottom margins

The top and bottom margins of a page limit the number of text lines or size of graphics you can place on a page. These margins are used for several things: header lines, footer lines, and page numbers. You will also find that readers also use them for their own handwritten notes.

A good basic rule is to use roughly 10% of the total page length as a top margin, and another 10% as the bottom margin. On an 11-inch long piece of paper, this gives about a one-inch margin at both the top and bottom of any page, leaving nine inches of room for text and graphics. If you use a standard line spacing of six lines per inch, this is 54 lines of text. That's quite a bit.

Another good reason for one-inch margins is the simple fact that most laser printers (such as the HP LaserJet family) cannot print in the top or bottom 1/2 inch of the page.

2.3 Left and right margins

The left and right margins of a page do a number of things: allow for the user to make notes; allow for change bars that indicate changes in the manual from one revision to the next; and provide enough space for the pages to be bound into some form of book (this is also called the 'binding width' in printing circles).

A good basic rule of thumb is to use 12% of the total page width for a left margin, and 12% for the right margin. Make sure that you allow enough room for the binding. Sometimes, it is best to allow a wider left margin on odd-numbered pages, and a wider right margin on even-numbered pages to allow for two sided pages to be bound properly. Many word processors will help you with this (*WordPerfect*, for example).

Using the 12% rule, an 8.5 by 11 inch page will have one inch side margins, leaving 6.5 inches for text. A standard 10-CPI (characters per inch) pitch then, allows 65 characters per line. (The HP LaserJet can't print in the left or right 1/2 inch of a page, either).

As you can see from these last two sections, the actual usable portion of a page using the basic rules is 6.5 inches wide by nine inches long. This allows for 3,510 characters per page (65 characters per line, 54 lines).

2.4 Columns

In a newspaper, you will find the text on a page partitioned into columns. Each paper has its own idea of how many columns to put on a page. Between the columns, you will find blank space called a "gutter".

The gutter width (in characters) is usually determined by counting the number of columns on a page and dividing by two. For example, if the page has four columns, you will find three gutters, each two characters wide. The column widths are adjusted to accommodate the gutters.

2.5 Rulers

Rulers define the horizontal positioning of text on the page. The tab positions are normally defined by the ruler, and they sometimes also define the left and right margins of the page.

Again, using the 8.5 by 11 inch paper, the typical ruler has the left margin set at column 10, the right at column 75 (giving one inch margins), and has tab stops at every fifth column (column 15, 20, 25 and so on). This allows good indention capabilities. Also, to facilitate centering, the center of a text line using this ruler is column 43.

2.6 Headers and footers

Headers are lines of text printed in the top margin of a page. Conversely, footers are printed in the bottom margin.

Usually, a good header is a single line which contains one of the following two things centered and boldfaced on it: the name of the document if this is an even numbered page; or the name of the chapter if this is an odd numbered page. It is placed in the center of the top margin. Using the one inch top margin, the header would be placed 1/2 inch from the top of the page.

A good footer is also typically a single line with the page number in boldface in it. The page number might be centered or at the left margin (even pages) or right (odd pages).

3. Titling and section numbering

Any reasonable document must be broken up into manageable chunks in order to be readable both as a text and as a reference volume. The way this is done varies greatly with the subject matter, so there are no rigid rules that can be stated. Nonetheless, here are some basic ideas.

Documents usually are segmented into sections which deal with one aspect or the other of the product being described. Depending on the complexity of the product, the document can be simply sectioned, or chapterized and sectioned.

If the document is of such complexity that chapterizing it makes sense (for example, if the document is going to be over 100 pages), then normally each chapter has a table of contents. The chapter is then partitioned into sections which deal with a particular aspect of that chapter.

3.1 Titling the sections

Determining the titles of chapters and sections can only be determined on a case by case basis by looking at the item you are attempting to write about, breaking it down into manageable pieces, and using a section of the document to describe each piece. The title used for the section should 'jump out' at you if you've broken it down properly.

In general, try to keep the title of the section short, perhaps no more than five words. Remember that the title of the section will appear in a table of contents and in the index of the document, so make it clear and concise. Titles are very much like the lines of latitude and longitude on a globe – they can tell you where you are and where you want to go.

I can remember a very catchy chapter title I saw once. It dealt with the programming aspects of a particular printer. The page divider (or 'tab') had the title 'Programming' on it, but the chapter title was 'Bells and Whistles'. Cute, but a bit obscure.

3.2 Section numbers

The way you number your sections depends on how you have broken down the document. This breakdown also commonly affects the page numbering scheme you use.

Normally, each part of a section number is separated from the other parts by use of a period or a hyphen. Which one to use is a personal choice, but use one in the section numbers and the other in the page number.

In a chapter/section numbering scheme, the first part of the section numbers is always the same – that of the chapter. The next part of the number (which is where we would start with a simple section numbering scheme) is the section number, starting at one, and increasing sequentially. If a section must be partitioned still further, additional numbers are allocated starting at one and increasing sequentially for each section above it.

For example, let's say a document was chapterized and that chapter six had two sections. The first section has two subsections while the second section has three sub-sections. The numbering scheme would appear thus:

```
6.   Chapter Title
6.1 First Section
6.1.1 First Sub-Section
6.1.2 Second Sub-Section
6.2 Second Section
6.2.1 First Sub-Section
6.2.2 Second Sub-Section
6.2.3 Third Sub-Section
```

The page numbering usually reflects the chapter number (if using a chapter/section scheme) or the section number followed by the page number of the chapter or section. For example, page 33 of the chapter described above would be "6-33".

4. Print enhancements

Special properties of text are used to add emphasis, clarify, or otherwise set off textual items. The most common ones are boldfacing, underlining, italics, special fonts, all-upper case text, and centering. Combinations of these are also common.

4.1 Boldfacing

Boldfacing text can be done fairly easily, and is extremely common. Basically, use boldfacing for: language elements in your text body (such as keywords, special characters, or syntactical elements); section numbers; sub-section headings; syntactical examples; computer output samples; listings of steps to perform or lists of items.

In general, when you wish to emphasize or clarify something (such as when you use a specific word as a keyword rather than its normal English meaning), use boldface. Emphasis also can be added by underlining, but that typically doesn't grab the reader's attention as well as boldface text.

4.2 Underlining

As mentioned before, underlining can add emphasis to text, most commonly in 'note sections'; for example:

```
NOTE:
This is a note section. Usually,
special information is given here.
```

It is often wise to boldface the text of a note section to draw attention to it.

Another good use of underlining is sample input to a program when you give examples of program prompts. Use boldface to show the prompt to the user, and underline the sample response. Here's an example:

```
What is your name? Rick
```

Here, the computer prompts with "What is your name?" and the response is "Rick".

4.3 Boldfacing and underlining

Use of both boldface and underlines really adds emphasis. This double technique is best applied in section headings.

If the document is chapterized and sectioned, then the section number (remember that this includes the chapter number as the first element) is printed in boldface, while the section title is both boldface and underlined. For an example, chapter six, section two and subsection three are shown below:

```
6.2 Section Title

6.2.3 Subsection Title
```

If the document is merely sectioned, then a similar technique is used, but the major section title is boldfaced and underlined:

```
6. Section Title

6.1 Subsection Title
```

Hopefully, that makes sense to you. It's a little difficult to explain. For a more full example of a sectioned document, look at the format of the one you're reading now!

4.4 Italics

Italics are normally used to print the title of other documents you are refering to, or to introduce a new keyword or phrase. Unfortunately, most printers can only do italics if you change a printwheel or change fonts. If that presents a problem, a good work around is to boldface the thing normally italicized and put it inside double quotes.

4.5 All upper case

Using all upper case letters (capitals) adds emphasis to text but is usually reserved for warnings to the reader. If you really wish to grab attention, use upper case, bolded, underlined text as in the example below:

```
NOW THAT WE HAVE YOUR ATTENTION...
```

Stand back, close your eyes, then open them. What did you notice first? Get it?

4.6 Block Centering

Block centering was covered in the section concerning underlining above. The note section is block centered text.

4.7 Justification

Most word processors and text formating utilities offer the capability of justifying your text. This means that the program will space out short lines so that the last character of each line lines up with the one above it. If your printer supports a mode called 'microjustification', text that is justified will look like a book.

If your program or printer can't support this mode, then the program will put enough spaces between words on a short line to justify the text. It often looks terrible.

There is a downside to justified text, even if it is microjustified - it can be difficult to read. How often have you read the same line in a book more than once, simply because there was no visual clue to move your eyes down to the next line?

If your paragraphs tend to be long, you might opt to turn off justification in your program, leaving the lines printed to different lengths. This produces 'ragged right' margins. It can make your text much easier to read.

5. Type Fonts

A *type font* is a particular way a character looks. Most typewriters use one of two fonts - Courier or Prestige Elite. Computer printers (especially laser printers) have a virtually unlimited range of different type fonts available.

Fonts are identified by several things: a name, a size, and letter spacing or *pitch*. The most common typewriter font is the Courier font, which has a size of eight points (there are 72 points to the inch, so a capital letter is 1/9th of an inch high) and a pitch of 10 characters per inch (also known as *pica* pitch). As a result, the 'usual' name for Courier is '8-point pica

Courier'. Most dot matrix printers use some permutation of this font. *Elite* pitch has 12 characters per inch, and is found less often.

Another fairly common pitch is *proportional* pitch. There is no fixed 'so many characters to the inch' setting for proportional fonts - each character takes up only as much space as necessary. For example, a capital W takes up the most space, while a lower case i takes up the least. Typeset publications such as books and newspapers are published in proportional fonts. Use of a proportional font makes the document look more 'polished' than a standard fixed-pitch font.

Another item relating to fonts is the number of lines per vertical inch allowed. A font with a small point size takes up less vertical space than the same font with a higher point size. Most printers default to six lines per inch.

If you are using a laser printer which has a large number of fonts available (and your word processor or desktop publishing package supports multiple fonts), you might make use of different fonts to emphasize text rather than using boldfacing, italics, or underlining. Or you might use them for your chapter and section titles. Whatever you decide, use them consistently. Don't go overboard and use different fonts just because you can. That makes your document look chopped up and lacking in unity. In extreme cases, people may mistake your hard work for a ransom note!

6. Graphic Images

There is an old adage that says 'A picture is worth a thousand words.' Through very careful study, it appears that a picture is really worth about 1523.274 words, give or take a bit. The idea here is that graphics are very valuable. When you are documenting something, pictures or graphics do several things:

- They break up the text into easier to read chunks
- They can describe what your text refers to
- They help clarify your text
- They take up lots of space!

For example, when describing how a screen looks, or where an important item is in a machine, by all means use a picture. Drawings are good to use if they aren't too cluttered. Photos are terrific, but if used, make sure that they have lots of contrast.

When placing your graphic image on a page, keep in mind some basic rules of thumb:

- If your graphic is half a page high or less, try to center it on the page vertically. You can put text above it and below it if you want.
- If your graphic is more than half a page high, put it on a page by itself.
- Caption your graphic. Use boldface. An example is:

```
Figure 1 - Front View of System
```

Also, make sure your graphic image is centered horizontally. Nothing looks tackier than a graphic image that is off center.

7. Table of Contents

A table of contents (TOC) is very important in any long document, especially one which is technical in nature. If your document is chapterized, it makes sense to have several TOCs - one for the whole document and one for each chapter. The one describing the entire document can be general in nature, giving the chapter title and a brief synopsis of what the chapter covers. More details would be found in the chapter's own TOC.

If the document is simply sectioned, a single TOC is all that's necessary. It would resemble a chapter TOC in a bigger document, since a sectioned document can be considered a single chapter document.

Major sections and subsections should be mentioned in the chapter TOC, along with the page numbers dealing with the subject. If you have room, perhaps a synopsis of the section might be in order. Use your own judgement.

8. Glossary

Any technical document which uses subject-related jargon or 'buzz words' should have a glossary where these terms are defined and described. Everyone has their own way of describing things, and not every one else necessarily understands that. In general, try to use 'industry standard' terminology whenever possible.

It must be said, however, that some things just aren't industry standard! A classic example is NASASpeak, a strange permutation of the English language created by the United States National Aeronautic and Space Administration. Listen to a press conference by these people and you'll understand what I mean! A more relevant example: video display terminals have been called CRTs, VDTs, VKTs, SPOs, consoles, terminals, and glass teletypes in the past. If someone not versed in the history of computers was told to refer to the VKT, he or she might not know what a VKT is. Remember, assume the user knows nothing about your internal terminology, and define your terms.

9. Indexing

An index is an indispensable tool in any technical manual or document. Every special item in the document should have a corresponding entry in the index listing the page numbers for all locations of that item.

Good practice for indexing can be summarized as:

- The items are sorted in alphabetical order.
- The page numbers referring to the item are entered.
- The page number where the item was either defined or first used (if not specifically defined) is entered in boldface type.

10. Editorial style

Much of what you will be writing about is pretty dry stuff. As most any programmer knows, software manuals are never much fun to read and often become downright boring and confusing. Here are some ideas to help avoid the doldrums.

Try not to write in a lecturing tone – no one likes to be talked down to. An occasional wry comment or humorous observation can make the document more interesting. Just don't overdo it: very few of us could make a living being standup comics.

Try to keep the narrative in the third person. For example, don't write "Then you push the button". Rather, write "Then the user would push the button." In other words, try not to use the word 'you' when referring to the reader. This makes the document more 'professional'.

Try to break your text up into easily understood paragraphs. Put a blank line between the paragraphs, and put two blank lines between the last line of a section and the title of the next section.

Make page breaks just before section titles whenever possible, and *never* allow a page break to occur in the middle of a paragraph. *This is a cardinal sin!*

Make use of graphics when necessary. They really help clarify and drive home certain points. Think how awkward it would be if your car's owner's manual had no pictures!

Never insult the intelligence of your audience! This can't be stressed enough. They're trying to learn from you, and treating them as imbeciles will accomplish nothing but make them hate the document, your product, and your company, and cause them to cast doubts upon your ancestry.

When dealing with technical subjects, it is often difficult to avoid using buzz words or jargon. Your reader is probably not as familiar with the subject as you are, so try to keep the jargon to a minimum. Provide a glossary for the jargon you do use, or define it when you use it the first time (see the section on fonts above for an example of this).

Let your document find its own length. Don't worry too much about how long it is, but try to be concise. Remember: "Orators usually make up in length what they lack in depth."

11. Conclusion

Try to make the manual an integral part of the product. That way, your customers are happy, your company is happy (because the reputation of the company is enhanced, which means more sales), and you are happy (because you won't have to answer questions all the time).

Good luck and Happy Typing!

GAME PROGRAMS WANTED!

Attention Developers and Freelance Programmers!

Virtual Reality Laboratories, Inc. is looking for a few sharp new entertainment game and educational game programs for the Amiga. To be accepted, these programs must be highly graphic with excellent sound capability. For a rapid review of your work send for guidelines to VRLI, Box 609, Atwood, CA 92601.

Do not submit until you have received and read our guidelines, but please do send for the guidelines TODAY!

This ad was prepared in PageStream (importing the bit-mapped image and the EPS logos and shears), and printed with PrintScript 1.1 at this size on the HP DeskJet.

PrintScript™ v1.1

The PostScript interpreter for the Amiga

PrintScript enables you to print PostScript files on your Preferences printer, giving you your printer's very finest work.

PrintScript 1.1 includes:

- A full Intuition interface- run PrintScript from either the Workbench or the CLI. You can choose your file with point-and-click requesters.
- More graphics support- rotate half-tone images and print Encapsulated PostScript clip art.
- Support for all major PostScript-generating word processing and desktop publishing packages, including the new releases.
- An AREXX port, and more!

PrintScript 1.1 also runs faster now, and in less memory.

Available soon... more PostScript fonts!

Pixelations

Pixelations, Inc.
P.O. Box 547
Northboro, MA 01532 USA
508-393-7866

All product names and symbols are the trademarks of their respective companies.

Cubic Spline Interpolation

Connecting the dots - smoothly!

by Eric Giguère
Copyright © 1989 Eric Giguère

Eric Giguère has been involved in the design of an ANSI-conformant C compiler at the Computer Systems Group at the University of Waterloo, where he is a computer science student in the co-op program.

Graphics programming often involves a tradeoff between space and time. Storing several bitmaps in memory and flipping between them is faster than calculating a new bitmap for each frame of animation, but requires a great deal more memory. It turns out that for many applications space is at a premium, and so we must turn to calculation-based methods to perform complex graphical manoeuvres.

Accuracy can be a problem. Calculations are often done using floating-point arithmetic, which is both slower and less accurate than comparable integer arithmetic. This type of accuracy can be improved with the addition of more precise floating-point libraries and/or a special floating-point coprocessor. There's almost always a tradeoff, however, in either time or money.

These accuracy problems are, for the most part, unavoidable to some degree because of the limitations common to all computers. There is a more important type of accuracy that we can tackle.

Interpolation

A typical problem in graphics is this: given a certain object, shape or form, we want to find the least number of data points that will allow us to redraw the shape with as much accuracy as possible without resorting to bitmaps. Using these data points we can *interpolate* the remaining points. This isn't a simple problem to solve, and there are many different ways to approach it. The method presented here is useful for outlines and similar graphical objects.

Say you are given a set of data points to plot as a two-dimensional graph. The data points are in (x,y) coordinate form and you are given a definite range for the x coordinates, but the corresponding y coordinate values are only known for the actual data points. Somehow you have to interpolate - guess - these values.

The simplest solution is to draw a straight line between adjacent points. (*Adjacent* here refers only to the x coordinates.) Thus given the points (2,5), (3,4) and (4,7) you'd draw a line from (2,5) to (3,4) and a second one from (3,4) to (4,7). The resulting graph will likely be very jagged. If you know, however, that the original data was plotted as one long, smooth curve, then the approximation you've just done is very inaccurate. The accuracy could be improved by using more known data points with smaller gaps between the x coordinates and hence drawing shorter lines, but again this requires more storage. There are also times when you simply don't have any more data points to use, in which case the straight-line approximation cannot be improved.

A *spline curve* is one way to get around this. A spline is a smooth curve that passes through a set number of data points. There are many types of splines. The ones we'll be discussing are referred to as *cubic splines*, the simplest ones. Most graphics applications use *B-splines*, but these are too complex to discuss here, and cubic splines will give you a feel for the use of splines in general.

Cubic spline interpolation

A cubic spline is made up of many smaller curves, one for each pair of adjacent data points. The spline is known as *cubic* because each of these smaller curves can be described using a cubic polynomial of the form:

$$ax^3 + bx^2 + cx + d$$

for the values of x between the two data points for that particular curve. The process is complicated by the fact that each data point (except the two endpoints) has two adjacent curves, and these curves must meet smoothly at that point. (In mathematical terms the first and second derivatives of the curves at that point must equal each other. See the references for more details.)

To explore cubic splines I created a module of routines that implement one of the spline algorithms, as well as a test program that demonstrates the use of these routines. The sources are listed at the end of this article and are available on the disk for this issue.

The CDSpline module

The header file *cdspline.h* must be included by all modules of a C program that use any of the functions in *cdspline.c*, the spline module. The header file defines an abstract data type *cspline* to hold the spline data, parts of which must be filled in by the user.

The module *cdspline.c* defines the following functions:

void NewDSpline(int n, cspline *sp)

Allocates enough storage to hold a spline with n data points.

void FreeDSpline(cspline *sp)

Frees the storage allocated by NewDSpline, allows sp to be used with different data points.

boolean SetDSpline(cspline *sp, int i, double x, double y)

Sets the i'th data point to the given (x,y) value.

void CalcDSpline(cspline *sp)

After the user has filled in the x and y arrays of the cspline type, this routine calculates the data for the spline interpolation.

double DSpline(double x, cspline *sp, int *err)

Given a pointer to a spline that has been calculated with CalcDSpline(), this routine returns the y value that interpolates the given x coordinate. Errors are indicated with the third parameter and defined in the header file.

There is another routine, **FastDSpline()**, which is faster than DSpline() but requires that the caller know the interval in which the x coordinate lies. See listing 2, *cdspline.c*, for complete documentation for the above functions.

Using the spline routines is fairly simple. First decide how many data points you will be using, and then call NewDSpline(), passing it a pointer to a cspline variable. The cspline data type is a structure that contains two arrays, x and y. Store the n data points into these arrays using the SetDSpline() routine. Then call CalcDSpline() to fill in the remainder of the structure with the data needed. Once this is done, call DSpline() to get an interpolated value for a given x value. Call FreeDSpline() when done with a spline.

Important: CalcDSpline() expects the x coordinates to be stored in *ascending* order in the cspline structure. It doesn't check to make sure they are, so it's up to you to sort them. Also, note that DSpline() (and FastDSpline()) can only be used with x values that are between the extremes of the data points. That is, if you calculate a spline using the points (1,2), (1.5,3) and (1.88,3.5), DSpline must be given an x value between 1 and 1.88, otherwise it will inform you of a range error.

The test program

The test program is a simple spline demonstration program that uses the routines listed above. The program opens a hi-res window on the Workbench screen. Use the mouse and the left mouse button to plot points in this window - these will serve as the data points for a spline. To see the spline, press the right mouse button. You can start a new spline now or press the right mouse button again to clear the screen. The window close gadget ends the program.

Run the program and play around with it to get a feel for splines. Note that it won't let you choose points to the left of any point you've already plotted - this is to ensure that the data points are sorted in ascending order. The plotted curves will always be smooth, except for parts that would otherwise be plotted outside the window.

Final notes

Splines are useful for approximating smooth objects using only a few data points. We've only touched the surface here. Most texts on numerical analysis will include a section on splines, as will books on computer graphics. My primary reference for this article was the third edition of *Numerical Analysis* by Richard Burden and Douglas Faires (Prindle, Weber & Schmidt, 1985), which includes algorithms for calculating splines. Most discussions about splines are heavily mathematical in nature, however, so be warned.

Listing 1: *cdspline.h*, header file for the cubic spline routines

```
/*
 * cdspline.h -- Type definitions for use with
 *               the (double) cubic spline routines
 */

#ifndef _CDSPLINE_H_

  #define _CDSPLINE_H_

  typedef struct cspline
    {
      short   n;      /* array dimension size */
      double *x,      /* known x-values */
             *y,      /* known y-values */
             *b,      /* filled in by CalcISpline */
             *c,
             *d;
    }
      cspline;

  typedef short boolean;
```

```
  #define SPLINE_OK          0
  #define SPLINE_RANGE_ERROR 1

  #ifndef NO_PROTOTYPES

    extern void    NewDSpline(), FreeDSpline(), CalcDSpline();
    extern double  DSpline(), FastDSpline();
    extern boolean SetDSpline();

  #endif

#endif
```

Listing 2: *cdspline.c*, cubic spline routines

```
/*
 * cdspline.c -- Routines for calculating a natural cubic spline
 *
 * Implementation by Eric Giguere
 *
 * This file is not Amiga-dependent and will compile on any system
 * supporting the calloc() and free() functions.
 *
 */

#include <stdio.h>

#define NO_PROTOTYPES

#include "cdspline.h"

#ifndef FALSE

    #define FALSE 0
    #define TRUE  1

#endif

extern void *calloc();
extern void  free();

/*
 * NewDSpline -- Allocate memory for the cspline data structure.
 *
 */

void NewDSpline( n, sp )
  int      n;
  cspline *sp;
  {
    sp->n = n;
    sp->x = calloc( n * 5, sizeof( double ) );

    if( sp->x == NULL )
      {
        puts( "NewDSpline: Could not allocate memory." );
        sp->n = 0;
        return;
      }

    sp->y = sp->x + n;
    sp->b = sp->y + n;
    sp->c = sp->b + n;
    sp->d = sp->c + n;
  }

/*
 * FreeDSpline -- Free memory used by a spline data structure.
 */

void FreeDSpline( sp )
  cspline *sp;
  {
    if( sp->x != NULL )
        free( sp->x );
    sp->n = 0;
    sp->x = NULL;
    sp->y = NULL;
    sp->b = NULL;
    sp->c = NULL;
    sp->d = NULL;
  }

/*
 * SetDSpline -- Set a data point for the given spline.
 *               The spline must first have been allocated with
 *               NewDSpline.  Returns FALSE if error occurs.
 *
 */

boolean SetDSpline( sp, i, x, y )
  cspline *sp;
  int     i;     /* data point # */
  double  x, y;  /* data point value */
  {
    if( i < 0 || i >= sp->n || sp->x == NULL )
        return( FALSE );

    sp->x[ i ] = x;
    sp->y[ i ] = y;

    return( TRUE );
  }

/*
 * The following function implements Algorithm 3.4
 * as found in Numerical Analysis, 3rd Edition, by Richard L. Burden
 * and J. Douglas Faires.  The algorithm calculates the cubic spline
 * coefficients given a set of data points (x,y).
 */

/*
 * CalcDSpline -- Calculate using doubles.
 *
 *   Sample calling sequence:
 *
 *      cspline sp;
 *
 *      --> allocate memory & fill in values for x & y
 *
 *         ......
 *
 *      CalcDSpline( &sp );
 *                   ^
 *                   |
 *                   -- spline data
 *
 */

void CalcDSpline( sp )
  cspline *sp;
  {
    register int n, i;
    static       cspline temp = { -1, NULL, NULL, NULL, NULL, NULL };

    /* Let's define some shorthand notations */

    #define h temp.x
    #define l temp.y
    #define m temp.b
    #define u temp.c
    #define z temp.d

    n = sp->n;

    if( n > temp.n )
      {
        FreeDSpline( &temp );
        NewDSpline( n, &temp );
```

```
    }

  for( i = 0; i < n-1; ++i )
      h[ i ] = sp->x[ i+1 ] - sp->x[ i ];

  for( i = 1; i < n-1; ++i )
    {
      m[ i ]  = sp->y[ i+1 ] * h[ i-1 ];
      m[ i ] -= sp->y[ i ] * ( sp->x[ i+1 ] - sp->x[ i-1 ] );
      m[ i ] += sp->y[ i-1 ] * h[ i ];
      m[ i ] *= 3.0;
      m[ i ] /= h[ i-1 ] * h[ i ];
    }

  l[ 0 ] = 1.0;
  u[ 0 ] = 0.0;
  z[ 0 ] = 0.0;

  for( i = 1; i < n-1; ++i )
    {
      l[ i ] = 2.0 * ( sp->x[ i+1 ] - sp->x[ i-1 ] ) - h[ i-1 ] * u[ i-1 ];
      u[ i ] = h[ i ] / l[ i ];
      z[ i ] = ( m[ i ] - h[ i-1 ] * z[ i-1 ] ) / l[ i ];
    }

  l[ n-1 ] = 1.0;
  z[ n-1 ] = 0.0;
  sp->c[ n-1 ] = 0.0;

  for( i = n-2; i >= 0; --i )
    {
      sp->c[ i ]  = z[ i ] - u[ i ] * sp->c[ i+1 ];
      sp->b[ i ]  = ( sp->y[ i+1 ] - sp->y[ i ] ) / h[ i ];
      sp->b[ i ] -= h[ i ] * ( sp->c[ i+1 ] + 2.0 * sp->c[ i ] ) / 3.0;
      sp->d[ i ]  = ( sp->c[ i+1 ] - sp->c[ i ] ) / ( 3.0 * h[ i ] );
    }

  return;

  /* Clear the macros */

  #undef h
  #undef l
  #undef m
  #undef u
  #undef z
}

/*
 * FastDSpline -- Given an x value and the interval in which
 *                it lies, return the corresponding y value.
 *
 */

double FastDSpline( x, sp, i )
  double   x;
  cspline *sp;
  int      i;
  {
    register double y, z;

    z = x - sp->x[ i ];

    y  = sp->d[ i ];
    y *= z;
    y += sp->c[ i ];
    y *= z;
    y += sp->b[ i ];
    y *= z;
    y += sp->y[ i ];

    return( y );
  }

/*
 * DSpline -- Given an x value, calculate the corresponding
 *            y value using the coefficients calculated by
 *            CalcDSpline.
 *
 *   Sample calling sequence:
 *
 *      cspline sp;
 *      int     err;
 *      double  x, y;
 *
 *      CalcDSpline( &sp );
 *
 *        ....
 *
 *      y = DSpline( x, &sp, &err );
 *                  ^   ^    ^
 *                  |   |    |
 *       x value -----  |    --- flag (see below)
 *                      |
 *                      ----- spline data
 *
 *   The flag returns the error condition:
 *
 *       SPLINE_OK          -- no errors
 *       SPLINE_RANGE_ERROR -- x value was out of range
 */

double DSpline( x, sp, err )
  double   x;
  cspline *sp;
  int     *err;
  {
    register int    i;
    register double y, z;

    if( x < sp->x[ 0 ] || x >= sp->x[ sp->n - 1 ] )
      {
        *err = SPLINE_RANGE_ERROR;
        return( 0.0 );
      }

    *err = SPLINE_OK;

    for( i = 0; i < sp->n - 2; ++i )
        if( sp->x[ i ] <= x && x < sp->x[ i+1 ] )
            break;

    return( FastDSpline( x, sp, i ) );
  }
```

Listing 3: *main.c*, test program for the spline routines

```
/*
 * Spline Test Program -- From the Transactor for the Amiga
 *                        by Eric Giguere
 *
 * File: main.c
 *
 *      Example of the use of the spline routines.
 */

#include <stdio.h>
#include "exec/types.h"
#include "intuition/intuition.h"
#include "cdspline.h"

#define MAX_POINTS    100  /* Max # points to accept */

#define XMIN            3  /* x,y coordinate limits for the window */
#define YMIN           10
#define XMAX          637
#define YMAX          198

#define FORE           2L  /* Foreground & background colours */
```

```
#define BACK          0L

struct IntuitionBase *IntuitionBase;
struct GfxBase       *GfxBase;
struct Window        *SplineWindow;

cspline               Spline;

extern void          *GetMsg();
extern void           ReplyMsg();
extern void           Wait();

boolean               ProcessEvent(), InRange();
void                  ProcessButton();

main()
  {
    struct IntuiMessage *msg;
    ULONG               event;
    boolean             done;
    int                 mouse_x, mouse_y;
    USHORT              code;
    double              x, y;

    Init();

    /* Allocate space for the spline data structure */

    NewDSpline( MAX_POINTS, &Spline );

    /* Wait & process events */

    done = FALSE;

    while( !done )
      {
        Wait( 1L << SplineWindow->UserPort->mp_SigBit );

        while( ( msg = GetMsg( SplineWindow->UserPort ) ) != NULL )
          {
            event   = msg->Class;
            mouse_x = msg->MouseX;
            mouse_y = msg->MouseY;
            code    = msg->Code;

            ReplyMsg( msg );

            if( ProcessEvent( event, code, mouse_x, mouse_y ) )
                done = TRUE;
          }
      }

    /* Free up memory */

    FreeDSpline( &Spline );
    CleanUp();
  }

/*
 * ProcessEvent -- Process an Intuition event
 */

boolean ProcessEvent( event, code, mx, my )
  ULONG  event;
  USHORT code;
  int    mx, my;
  {
    switch( event )
      {
        case CLOSEWINDOW:
            return( TRUE );
        case MOUSEBUTTONS:
            ProcessButton( code, mx, my );
            break;
      }

    return( FALSE );
  }

/*
 * ProcessButton -- Process a mouse button event
 */

void ProcessButton( code, mx, my )
  USHORT code;
  int    mx, my;
  {
    register int  i, y;
    static int    next = 0, last_x = 0;
    double        x;

    switch( code )
      {
        case SELECTDOWN: /* Add a new point of spline data */

            /* Make sure it's in range */

            if( next < MAX_POINTS - 1 && mx > last_x &&
                InRange( mx, my ) == TRUE )
              {
                SetDSpline( &Spline, next, (double) mx, (double) my );
                ++next;
                last_x = mx;
                Move( SplineWindow->RPort, (long) mx, (long) my );
                Draw( SplineWindow->RPort, (long) mx, (long) my );
              }

            break;

        case MENUDOWN: /* Plot spline OR clear the screen */

            if( next > 0 ) /* Plot spline */
              {
                /* Calculate the spline */

                Spline.n = next;        /* # of data points */
                CalcDSpline( &Spline ); /* calculate */

                /* Plot using fast spline routine --- normally
                   the DSpline routine should be used */

                Move( SplineWindow->RPort, (long) Spline.x[0],
                      (long) Spline.y[0] );

                for( i = 0; i < Spline.n - 1; ++i )
                  {
                    for( x = Spline.x[i]; x < Spline.x[i+1]; x += 1.0 )
                      {
                        y = (int) FastDSpline( x, &Spline, i );

                        if( InRange( XMIN, y ) )
                          {
                            Draw( SplineWindow->RPort, (long) x, (long) y );
                          }
                      }
                  }

                /* Reset */

                next     = 0;
                last_x   = 0;
                Spline.n = MAX_POINTS;
              }
            else /* Clear screen */
              {
                SetAPen( SplineWindow->RPort, BACK );
                RectFill( SplineWindow->RPort, (long) XMIN, (long) YMIN,
                          (long) XMAX, (long) YMAX );
```

```
            SetAPen( SplineWindow->RPort, FORE );
          }

        break;
      }
  }

/*
 * Check to see if a given point is in range
 */

boolean InRange( x, y )
  int x, y;
  {
    if( x >= XMIN && x <= XMAX && y >= YMIN && y <= YMAX )
        return( TRUE );

    return( FALSE );
  }
```

Listing 4: *init.c*, routines for test program

```
/*
 * Spline Test Program -- From the Transactor for the Amiga
 *                     by Eric Giguere
 *
 * File: init.c
 *
 *      Sets up the hi-res screen.
 */

#include <stdio.h>
#include "exec/types.h"
#include "intuition/intuition.h"

/*
 * The window -- uses the Workbench screen
 */

static struct NewWindow NWin =
  {
    0, 0, 640, 200,
    0, 1,
    MOUSEBUTTONS | MOUSEMOVE | CLOSEWINDOW,
    ACTIVATE | REPORTMOUSE | SMART_REFRESH |
    WINDOWCLOSE | RMBTRAP,
    NULL,
    NULL,
    (UBYTE *) "Cubic Spline Plotter",
    NULL,
    NULL,
    0, 0, 0, 0,
    WBENCHSCREEN
  };

/*
 * The colour table we'll use
 */

static UWORD NewCols[ 4 ] =
  {
    0x0000, 0x0fff, 0x0ed0, 0x07df
  };

/*
 * Storage for the old colour tables
 */

static UWORD OldCols[ 4 ];

static struct IOStdReq       SplineMsg;

extern struct IntuitionBase *IntuitionBase;
extern struct GfxBase       *GfxBase;
extern struct Window        *SplineWindow;

extern UWORD                GetRGB4();
extern void                 LoadRGB4();
extern void                 *OpenLibrary(), CloseLibrary();
extern void                 *OpenWindow(), CloseWindow();

/*
 * Set up the window
 */

void Init()
  {
    struct ViewPort *vp;
    int             i;

    /* Open libraries */

    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary( "intuition.library", 0L );

    if( IntuitionBase == NULL )
      {
        puts( "Could not open Intuition library." );
        exit( 0 );
      }

    GfxBase = (struct GfxBase *)
        OpenLibrary( "graphics.library", 0L );

    if( GfxBase == NULL )
      {
        puts( "Could not open graphics library." );
        exit( 0 );
      }

    /* Open window */

    SplineWindow = OpenWindow( &NWin );

    if( SplineWindow == NULL )
      {
        puts( "Could not open window." );
        exit( 0 );
      }

    /* Load new colours */

    vp = &SplineWindow->WScreen->ViewPort;

    for( i = 0; i < 4; ++i )
        OldCols[ i ] = GetRGB4( vp->ColorMap, (long) i );

    LoadRGB4( vp, NewCols, 4L );
    SetAPen( SplineWindow->RPort, 2L );
  }

/*
 * Clean up
 */

void CleanUp()
  {
    /* Reset colours */

    LoadRGB4( &SplineWindow->WScreen->ViewPort, OldCols, 4L );

    /* Close everything */

    CloseWindow( SplineWindow );
    CloseLibrary( GfxBase );
    CloseLibrary( IntuitionBase );
  }
```

Debugging With MetaScope

A tutorial introduction

by James C. Lawrie

After having spent a lot of time thinking about how to write a program, double-checking your logic as you type it in, you somehow feel entitled to have your program run right the first time. Yet, how many times does this happen? If you're like me, not very often. Welcome to the world of debugging.

Many programmers track down bugs by inserting debugging statements into the code. These often consist of calls to the function printf() in C, to list to the screen the state of variables suspected of being involved in the bug. Other languages have their own debugging statements, but the purpose is the same: to determine which variables have gone astray.

This approach is sufficient for small projects, and even large ones for a very patient person – one who is willing to recompile his or her program every time debugging code is changed.

Introducing Metascope

However, there are specialized tools, called *debuggers*, that provide much more sophisticated capabilities than strategically placed printf() calls. Furthermore, they permit 'on the fly' changes to the target program's code so that you can modify its operation interactively, without recompilation.

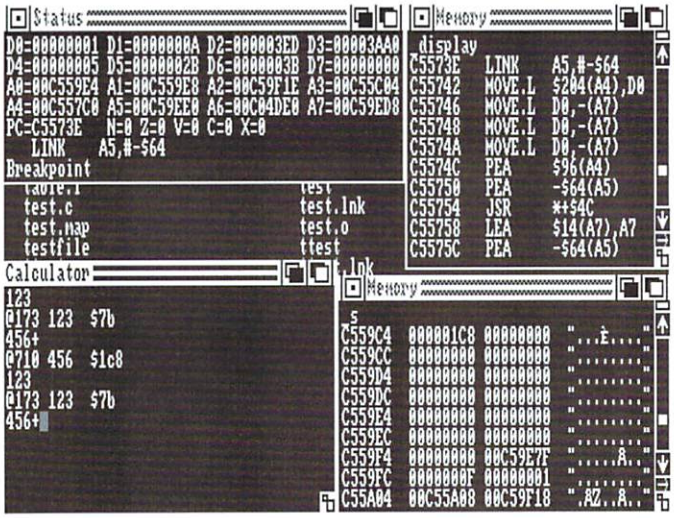


***Figure 1**: Metascope's windows reveal register contents, variable values and program output.*

The debugger I use with my Amiga, *MetaScope*, allows me to look at any location in memory and change its contents. I can also examine and modify the values in the registers. I can step through the program one instruction at a time; or set breakpoints where I want the program to stop so that I can look at data in my program. MetaScope opens its own windows so that debugging does not interfere with the input or output of my program. For this article, we will only look at a small subset of what MetaScope can do.

MetaScope can be run by itself, or with a program to be debugged. By itself, you can poke around memory looking at the innards of the Amiga software, and possibly crash the Amiga if you change something important to AmigaDOS. To examine a program, you can either load it from the *Project* menu or, when you first run MetaScope from the CLI, specify the program name and its arguments as arguments for the Metascope program; for example:

```
MetaScope MyProgram Arg1 Arg2
```

If you now give the **Go** command, *MyProgram* would behave just as if you had run it straight from the CLI – except that you have the MetaScope *Status* window on the screen.

Let's look at some of the things MetaScope allows you to do. The first item worthy of mention is the aforementioned Status window, initially positioned in the upper left hand corner of the screen. Displayed in it are the contents of the 68000's registers, the value contained in the program counter, the status register condition codes, the disassembled instruction pointed to by the program counter, and the current state of the program: whether it is executing, stopped or in some other state.

Another window available is the *Memory* window. This shows a section of memory in any of a number of formats, including disassembled instruction format and the default, hexadecimal format. Symbolic address information is included if it is available to MetaScope. More than one Memory window can be opened at the same time, allowing you to look at more than one section of memory at a time.

The base address for the Memory window can be set to be either 'static' or 'dynamic'. The former fixes the base address

to the value of an expression you enter and displays memory beginning at that address. The latter causes the debugger to re-evaluate the expression every time your program exits to MetaScope. Thus, if you enter *PC* as the expression for the Memory window base address, each time you hit a breakpoint, or stop the program, the Memory window will be displaying the area of code to be executed next. This can be very useful when you wish to look at not just the next instruction to be executed, as is displayed in the Status window, but the several instructions following.

One technique I have found useful for debugging is to have two Memory windows open. One displays the code I am about to execute, as just described; the other window displays the stack frame currently active in the program by dynamically setting the base address to address register A7, or the stack pointer. When debugging C programs, this display shows the local variables, followed in order by the old frame pointer, the return address of the function and the arguments passed to this function by the calling routine. We will see an example of this later in the article.

Another type of window, the *Breakpoint* window, lists the addresses of the currently set breakpoints; the number of times the program has come through the breakpoint; the number of times the program must come through the breakpoint before program execution actually breaks; and the conditional expression, if one exists. The conditional expression controls the triggering of the breakpoint.

Where to start debugging

Before we get down to business and show MetaScope in action, let's talk about debugging. Where do you start? How do you decide what to look at? And how do you know a problem when you see it?

That last question may seem ridiculous, but I believe one of the toughest parts of debugging is recognizing a problem in the making. The cause of the bug in a subroutine may be an incorrect value in a register copied from a variable before another subroutine had a chance to put the correct value in the variable, because yet another variable was set incorrectly, causing the first subroutine to run before its time. Recognizing the problem as being a lack of synchronization between the two subroutines caused by a design error, and not just an incorrect variable value, takes experience.

I always like to begin debugging at the point where it becomes obvious that the program is not performing as it should, then backtrack as far as is needed to uncover the root of the problem.

Let's take a simple example. Suppose we are writing a program which calls the C library function printf() to perform some output for us. Perhaps this is a disassembler and we are displaying the mnemonic of an instruction; one of the arguments to the printf() function, in this case, would be a string selected from an array of strings. Somewhere inside of printf(), we crash.

Did printf() all of a sudden stop working? Not likely. But this is the place we should start looking for the bug – not because we believe printf() to be in error, but because that is where the problem first became apparent.

We set a breakpoint inside the printf() code to see what is being passed for arguments. Upon hitting the breakpoint, we discover that the address of the string we should be printing is not what it should be.

Now is the time to see what called printf() to do the output. We backtrack to that code and we find the offender. It turns out that, although we were trying to pass a string address from an array of strings, we had mistakenly declared that array to consist of pointers to *char*, rather than pointers to pointers to *char*.

The point of all this is that even though you may discover an error in previously debugged code, or code you are sure couldn't be wrong, don't discount the error. The old adage, 'garbage in, garbage out', holds particularly true here. You can cause a good routine to behave very badly by giving it a bad argument, such as an invalid pointer.

But don't look elsewhere first, for the same reason. For even though printf() fouled up and you were sure it couldn't, that *is* the place to start to look, for that is the first place the error showed up. Later you can backtrack to where the bad pointer is getting passed to printf().

Stalking a real live critter

Now let's look at how MetaScope can help us. The example I am going to show is a real-life bug. I had coded a simple multi-radix (multiple number base) integer calculator, intending to insert a deliberate bug. Instead, the bugs were already there. Let's find one of these bugs, employing the principles discussed above.

This calculator is designed to accept numeric arguments of binary, decimal, octal or hexadecimal type, and the four fundamental arithmetic operations. Decimal numbers are entered as decimal digits. Binary numbers are entered by preceeding the binary digits with a %, octal numbers are preceded with @, and hexadecimal numbers are preceded with $. Hexadecimal digits A through F may be entered in either upper or lower case. The calculator also accepts lower case c for clearing the current entry, or upper case C for clearing the entire calculation.

The calculator uses reverse Polish notation, which I have found to be much easier to use than algebraic entry. To perform an addition operation, for example, you would enter one of the numbers to be added, then strike either the **Return** key or the **Enter** key to push the number onto the calculator stack. The second number would then be entered and the operation key + struck. The answer should then appear and the calcula-

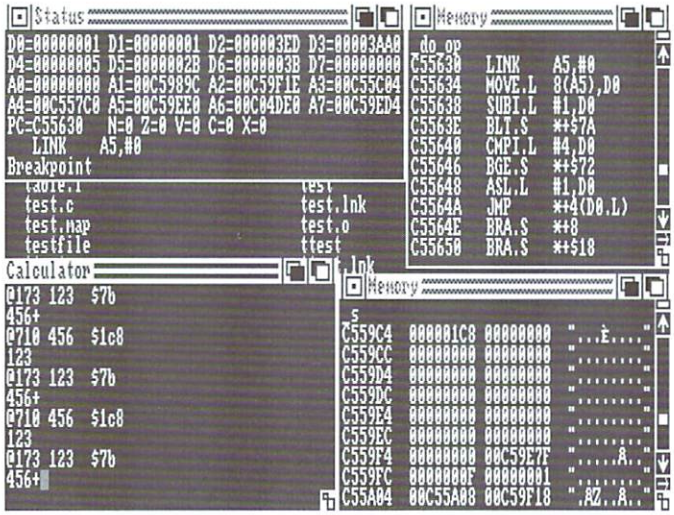


Figure 2: Checking the calculator's operand stack.

tor would be ready for more numbers or operations to be entered.

However, with the program as shown in Listing 1, if we try to perform an addition with two numbers, say 123 and 456, we do not get 579, as expected. We get 456. Trying the multiplication or division operations, we get 0 as the answer. Subtraction gives us -456. We have a bug.

Using one of our principles, that of going to the first place we know the error occurs and working backwards, takes us to the display() routine. We will put a breakpoint at the entry to display() and look at what is on top of our calculator operand stack. Listing 1 shows us that display() is supposed to list out the first entry in the operand stack in octal, decimal and hexadecimal format.

We invoke the program with MetaScope in control. This loads the calculator program and opens a Status window in the upper left part of the screen. Then we open a Memory window and

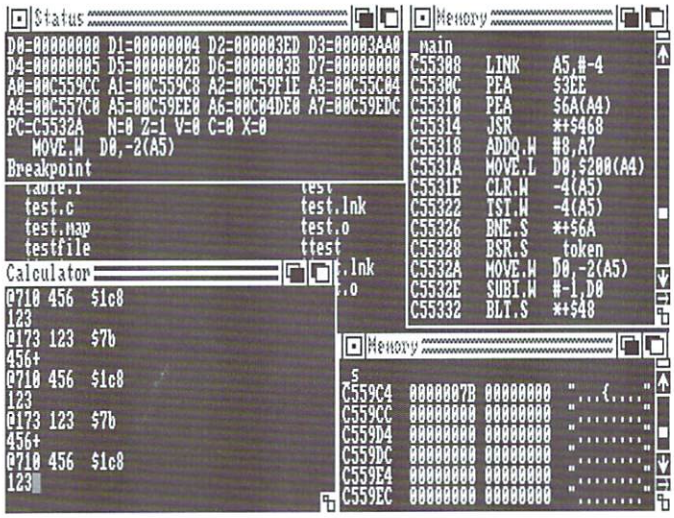


Figure 3: What's token() passing to main()?

give as its base address the variable name *s*. This must be entered as *_s* because the Lattice C compiler I use prefixes all external names with the _ character. Likewise, we set a breakpoint at the beginning of display() using the symbolic name *_display*. Then we execute the **Go** command.

A window opens in the lower left part of the screen with the title *Calculator*. As you enter the numbers and strike the **Enter** key, you hit the breakpoint a couple of times. This is because the program displays the numbers as they are entered. Use the **Go** command to continue program execution from the breakpoint. You actually want to look at the breakpoint executed after typing in the +, for it is here that the addition has been completed and the result is displayed.

Figure 1 shows us stopped at this breakpoint. Another Memory window has been opened that shows the code in the display() routine. Notice that in the lower memory window, *_s* has as its first entry the longword $1C8, which is 456 decimal. This tells us that the problem occurred before the display() subroutine was invoked, because it has correctly listed the top value, 456, on the calculator operand stack.

So we must back up further to discover the problem. A good guess would be that we are either not performing the addition operation, or performing it incorrectly. After all, subtraction gives the negative of the last number entered, and multiplication and division both give 0 as the result. So we set a breakpoint in the do_op() function, hoping to step through and uncover why the operation is not being performed correctly.

Figure 2 shows the screen as it looks when we hit the breakpoint at the beginning of the do_op() function. Before going any further, we look for our operands on the calculator stack. We should find both the 123, or $7B, and the 456, or $1C8.

Only the hex $1C8 is on the operand stack! Do_op() uses both the first and the second entries on the operand stack and returns the result to the stack. But it can't operate correctly if the operands are wrong. This is our 'garbage in'. Performing the addition operation produces the 'garbage out'.

We need to back up another level. But to where? What could have put the operands on the calculator stack incorrectly? The best way to find out is to go back to a place in the program where the operands get put on the calculator stack. The routine token() does this just prior to returning. We could breakpoint there, and watch what happens, or go back to main(), where token() is called, and monitor the calculator stack at that point.

I chose to look at the stack from the main() routine, simply so I could verify that all tokens, including the operators, were being parsed correctly. A quick look at the program shows that only numbers go to the end of the token() routine. Other types of tokens, such as the **Enter** key, are returned earlier. Figure 3 shows that the breakpoint is on the instruction just after the call to token(), in main(). What is occurring here is that data register D0 is being stored in a local variable. D0 is the token()

function's return value; if you look at the token() routine, you'll see that the last statement is a *return(0)*. (A function's return value always comes back in D0 with the Lattice compiler.) In the program, this tells the calling program that a numeric argument was parsed and put on the calculator stack.

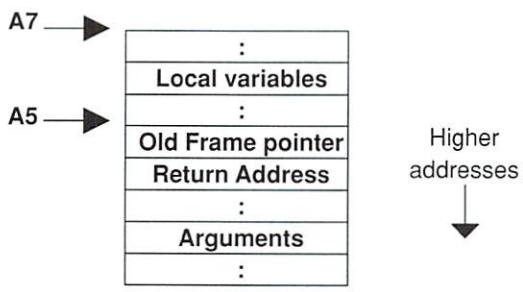
If you then restart the calculator program using the same operands, and look at its operand stack after each breakpoint, you will see the 123 appear on the operand stack, followed by the **Enter** key being returned. Then the 456 will be put on the stack. But it appears that the 456 wiped out the 123. The 123 is supposed to be pushed up to the second position on the operand stack before the 456 gets put on the stack. When the **Enter** key is returned to main(), the entire stack is lifted, or pushed, via a call to lift(). It must be that this routine is not functioning properly.

Setting the trap

Now it is time to go to lift() and look at what is occurring before there. Listing 2 shows part of the output of the *Object Module Disassembler* that comes with the Lattice compiler. The source to lift() is shown interspersed with the assembly instructions generated by the compiler. These instructions perform the operations of the C statements they follow in the listing.

Before continuing with the debug, let me digress for a moment to discuss the program stack (not to be confused with the calculator program's operand stack) conventions used by the Lattice compiler. I will refer to Listing 2, and also to Figure 4, which shows the code of lift() as it appears in memory. The other Memory window shows part of the 68000 stack, as pointed to by the stack pointer in address register A7 and the frame pointer in address register A5. A5 is actually two greater that A7, due to the **LINK A5,#-2** instruction at the beginning of the routine.

The **LINK** instruction does is creates a stack frame (hence the term 'frame pointer') for the currently executing routine. The contents of the register specified in the instruction is pushed onto the stack, and the register is updated with a pointer to the old contents just pushed onto the stack. Then the stack pointer is decremented by the size of the frame, also given in the **LINK** instruction. This frame pointer acts as a reference to arguments passed into the routine, to local variables used by the routine, and to information needed to return from this routine. A frame looks like this in memory (with addresses increasing down the page):



The frame pointer, A5, points to the location in the stack containing the old frame pointer, which was put there when the **LINK** instruction executed.

Just below the old frame pointer is the return address, pointing to the instruction after the instruction which called this routine. Remember, stacks in the 68000 grow down, so that data below

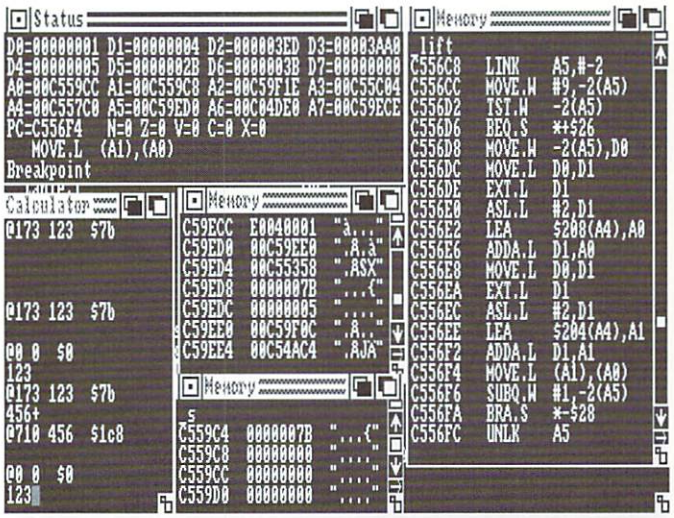


***Figure 4**: The compiler created a frame of two bytes, for the lift() function to hold the local variable* i.

the top of the stack is at a higher memory address. As shown in Figure 4, the longword beginning four bytes beyond where A5 points is the return address, $C55358. The lift() routine was called from main(), and if you were to open a memory window to the main() routine, you would find that this return address points to the instruction following the **JSR** to _lift.

Below that are any arguments to the routine. They would begin eight bytes beyond where the frame pointer points. Lift() does not have any arguments.

For lift() the compiler created a frame of two bytes, which correspond to the local variable, *short i*. Its offset is -2 from the frame pointer, or $FFFE(A5).

Notice that in Listing 2, the first instruction after the **LINK** is a **MOVE.W** to this memory location. The compiler generated an instruction which moves a word operand to *i*, in this case a nine. The nine comes from the macro MAX_SIZE-1, which is the number of operands in the calculator stack minus one.

Every call to a routine in C will do some similar processing. The frame pointer need not be in A5 – any address register except A7 will do. Lattice just happens to use A5 because A6 is used for other purposes.

Notice in Figure 4 that we have breakpointed on a **MOVE** instruction. We are shifting one operand of the calculator stack up to another position – lifting it up, as the function name suggests. You may verify that the addresses shown do indeed

point to two entries in the calculator stack. Notice that the entry being moved, pointed to by A1, corresponds to *s[1]*. It is being moved to *s[2]*.

Now watch what happens.

After the **MOVE** instruction, we subtract one from our local variable *i* and go to test it. Figure 5 shows us about to take the branch instruction that gets us out of the loop before we have had a chance to push the top operand up one position. Notice that the Z flag is set in the Status window, which indicates that a **BEQ** instruction will indeed branch. It looks like we are exiting the lift loop one entry too soon.

To solve this problem, we must change the lift() routine. Instead of simply testing that *i* is non-zero in the *for* statement, we should check that *i* is greater than *or equal to* 0. This will allow the loop to proceed once more and lift the first entry into the second position.

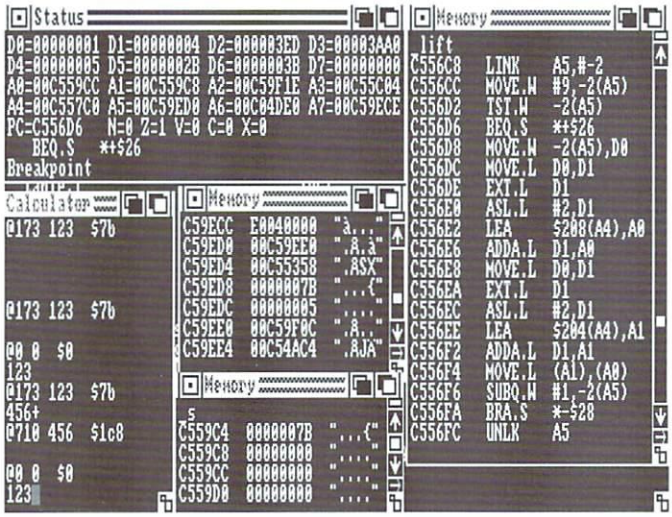


***Figure 5**: Got it! The **BEQ** instruction forces an exit from the loop in lift() before the operand is lifted up the stack.*

We could exit the program here, edit the source and recompile. But MetaScope provides an easier way to verify our solution: change the code in memory!

We can change the **BEQ** instruction to a **BLT**, to allow the loop to proceed one more time. Double clicking on the **BEQ** instruction opens a window and prompts for a new instruction. This is referred to as 'patching' a program.

Removing all the breakpoints and letting the program continue, we see that, indeed, the addition of 123 and 456 produces 579. Subtraction, multiplication and division now work as well.

Debugging can sometimes be a frustrating experience. However, with tools like MetaScope and some creative insight, debugging programs can actually be as much fun as writing them.

Listing 1: *Calc.c*

```
/* C program example.  A calculator with binary, octal, decimal, hex */

#include <stdio.h>

#define NEW 1006          /* AmigaDOS file attribute */
#define MAX_SIZE 10       /* Maximum size of stack */
#define PLUS 1            /* Addition */
#define MINUS 2           /* Subtraction */
#define STAR 3            /* Multiplication */
#define SLASH 4           /* Division */
#define ENTER 5           /* The ENTER button */
#define DISPLAY 6         /* Used to flag display of s[0] */
#define EOF_KEY 0x1c      /* End-of-file key */

char reread = 0;          /* Character to reread */
char lift_pend = 0;       /* Flag to enable lift of stack */
long window;              /* Window handle */
long s[MAX_SIZE];         /* The stack */

short token(), do_op(int);
void main(int, char **), lift(), drop(long), display();

void main(ac, av)
int ac;
char *av[];
{
    short op;             /* Operation variable */
    short exit;           /* Exit flag */

    window = Open("RAW:0/100/320/100/Calculator",NEW);

/* Forever loop, broken by end-of-file */

    for (exit=0; !exit;) {
        op = token();
        switch (op) {
            case 0:       /* Operand parsed */
                break;
            case ENTER:
                lift();
                display();
                break;
            case PLUS:
            case MINUS:
            case STAR:
            case SLASH:
                do_op(op);
            case DISPLAY:
                display();
                break;
            case -1:      /* EOF or serious error */
                exit++;
                break;
            default:      /* Parsing error */
                Write(window,"?? Error ??\n", 12);
        }
    }
    Close(window);
}

/* Parse a token and return a type indicator. Also, if the token is an operand,
   return the value of the operand to the place pointed to by the argument.
*/
short token()
{
    char c;
    unsigned short m, i;
    unsigned short radix = 10;
    unsigned long n = 0;

    if (reread) {
        c = reread;
        reread = 0;
    } else {
        if (Read(window, &c, 1) <= 0)
```

```
            return(-1);
    }
    switch (c) {
        case '\r':
            return(ENTER);
        case EOF_KEY:
            return(-1);
        case 'c':
            s[0] = 0;
            return(DISPLAY);
        case 'C':
            for (i=0; i<MAX_SIZE; i++)
                s[i] = 0;
            return(DISPLAY);
        case '+':
            Write(window,&c,1);
            return(PLUS);
        case '-':
            Write(window,&c,1);
            return(MINUS);
        case '*':
            Write(window,&c,1);
            return(STAR);
        case '/':
            Write(window,&c,1);
            return(SLASH);
        case '%':
            Write(window,&c,1);
            radix = 2;
            break;
        case '@':
            Write(window,&c,1);
            radix = 8;
            break;
        case '$':
            Write(window,&c,1);
            radix = 16;
            break;
        default:
            if (c >= '0' && c <= '9')
                break;
            return(-2);
    }
    if (radix != 10)
        if (Read(window, &c, 1) <= 0)
            return(-1);
    for (;;) {
        if (c >= 'a' && c <= 'z')
            m = c-'0'-0x20;
        else
            m = c-'0';
        if (m >= (9+'A'-'9'-1))
            m -= 'A'-'9'-1;
        if (m < 0 || m >= radix)
            break;
        n = n*radix+m;
        Write(window,&c,1);
        if (Read(window, &c, 1) <= 0)
            return(-1);
    }
    if (lift_pend) {
        lift_pend = 0;
        lift();
    }
    s[0] = n;
    reread = c;
    return(0);
}

/* Perform the operation */
short do_op(op)
{
    switch (op) {
        case PLUS:
            drop(s[1] + s[0]);
            break;
        case MINUS:
            drop(s[1] - s[0]);
            break;
        case STAR:
            drop(s[1] * s[0]);
            break;
        case SLASH:
            if (s[0] == 0) {
                drop(s[1]);
                return(0);
            }
            drop(s[1] / s[0]);
            break;
    }
    lift_pend++;
    return(1);
}

/* Lift the stack */
void lift()
{
    short i;

    for (i=MAX_SIZE-1; i; --i)
        s[i+1] = s[i];
}

/* Drop the stack and place a new operand onto it */
void drop(op)
long op;
{
    short i;

    s[0] = op;
    for (i=2; i<MAX_SIZE; i++)
        s[i-1] = s[i];
}

/* Display s[0] */
void display()
{
    char buf[100];

    sprintf(buf,"\n@%o %d  $%x\n",s[0],s[0],s[0]);
    Write(window,buf,strlen(buf));
}
```

Listing 2: Output of lift() by *Object Module Disassembler*

```
/* Lift the stack */
void lift()
{
03C0 4E55FFFE                LINK      A5,FFFE
 short i;

 for (i=MAX_SIZE-1; i; --i)
03C4 3B7C0009FFFE            MOVE.W    #0009,FFFE(A5)
03CA 4A6DFFFE                TST.W     FFFE(A5)
03CE 6724                    BEQ       000003F4
  s[i+1] = s[i];
03D0 302DFFFE                MOVE.W    FFFE(A5),D0
03D4 2200                    MOVE.L    D0,D1
03D6 48C1                    EXT.L     D1
03D8 E581                    ASL.L     #2,D1
03DA 41EC 0008-02.2          LEA       02.00000008(A4),A0
03DE D1C1                    ADDA.L    D1,A0
03E0 2200                    MOVE.L    D0,D1
03E2 48C1                    EXT.L     D1
03E4 E581                    ASL.L     #2,D1
03E6 43EC 0004-02.2          LEA       02.00000004(A4),A1
03EA D3C1                    ADDA.L    D1,A1
03EC 2091                    MOVE.L    (A1),(A0)
03EE 536DFFFE                SUBQ.W    #1,FFFE(A5)
03F2 60D6                    BRA       000003CA
}
03F4 4E5D                    UNLK      A5
03F6 4E75                    RTS
```

Hard Disk Husbandry

Making the most of those megabytes

by Jay Gross

Okay, your wallet is thinned down a bit, and your new hard disk drive is purring along next to (or inside of) your Amiga. What's next? – the Harddisk Attitude Adjustment period. This is when your fingers learn to stop automatically typing, and you drag out the AmigaDOS books again to figure out a bit more about the Path, Assign, and Cd commands. Here are some simple hints to help you get your money's worth out of your hard disk investment.

To partition or not to partition

Before you load up your hard disk, you have to make the dreaded partition decision. Do you format your giant volume as one big drive or as two, three, or more partitions? And what is a *partition*, anyway?

First, the easy part. Basically, a partition is a chunk of space on a hard disk drive that is treated by the Amiga as though it were a separate hardware device. Although the names of partitions are up to the user, the Amiga convention would be dh0:, dh1:, dh2: and so on, so that's what we'll call them here. The Amiga will know those 'drives' by their volume names, too, just as the disk in df0: can be accessed by either 'df0:ProgramName' or 'VolumeName:ProgramName'.

However, just as adding another floppy disk drive takes some system memory (CHIP memory!), so does mounting multiple partitions of a hard disk drive. Each partition or drive takes about 20 Kb; some controller software consumes more. If you put four partitions on your harddrive, for example, there went 80 Kb! Then, if you did something that really wanted all the CHIP memory it could get, you would likely receive a visit from the friendly Amiga Guru.

So, partitions cost CHIP memory. That's most of the 'Why not...' to partition. Now the 'Why...'.

When the computer crashes or is rebooted while writing a file, AmigaDOS will want to run the disk validator on any volume you were writing to when the interruption occurred. On a very large volume, the validator can take many minutes to do its thing, during which time retrieving files things from the disk is slowed way down.

A particularly bad crash might force you to reformat and reload a large volume. With multiple partitions, reformatting would be necessary only on the partitions that developed the problem.

The summary of this is, if you do things that are subject to gurus and lockups or if you're developing something that tinkers with AmigaDOS files and file structures, you should expend whatever memory it costs to have a partition set aside for your experiments.

Another advantage of multiple partitions is that the directory trees can be kept under control. However, the AmigaDOS Assign command can be used for the same purpose at far less memory cost: just make a directory and Assign a name to it as though it were a volume (it won't have a disk icon, however).

Then there's the compromise approach to partitioning. You create several volumes, but Mount only the ones you need. You could, for example, have separate partitions for software development, program testing, WorkBench, and desktop publishing. Since the desktop publishing software will not like coping with decreased CHIP memory, you could create a boot disk for it that omits the Mount command for the other partitions. Then, after doing your work in desktop publishing, you could Mount any of the other partitions you needed, without having to reboot the computer.

The Mount command works any time from the command line, not just in the startup-sequence. Unfortunately, AmigaDOS doesn't have an *Un*-Mount option. Under 1.3 you can re-use device names with the Assign Remove command, which will remove anything in the system that AmigaDOS knows about.

Ready for an experiment? Ready for a Guru? If you don't mind rebooting your system, try this:

```
Assign con: remove
```

Wasn't *that* festive? That command gets rid of the console device, which means the keyboard can't talk to the Amiga. The computer then *insists* on mouse input. There's probably a commercially useful application of this trick somewhere.

Game for another one? This one will make a big mess, too, so watch out.

```
Assign df1: remove
```

After you issue this command, the WorkBench or the volume name is the only thing that will work with df1:. You cannot access device df1: from the CLI; the computer will claim it doesn't exist, even though it will honour commands that address the disk in df1: by its volume name.

After using the above command, you can then Assign the df1: device anywhere you like. You could, for example, call your hard disk df1: and have a df1: that works at hard disk speed.

Alas, while the Assign Remove command frees up the device name for re-use, it doesn't free the device buffers. You can Assign Remove a partition after you're through with it, but you can't get its memory back.

If you don't mind rebooting, you can use several different partition configurations by creating boot disks tailored to different applications. Think carefully about the partitioning question before you start, though; changing the partitioning of a hard disk requires reformatting.

Subdirectories grow on trees

The first and most important thing to put on your hard disk is the Amiga WorkBench. You want the computer to look on the hard disk first, even if you're not indulging in that lazy legacy, autobooting. To get your hard disk to act as though it booted the system when it didn't, you need to convince it that its copies of the system files (found in libs:, devs: and l:) are the ones to check first. A simple Assign command in your startup-sequence will do that, but put the command up near the top, so control can be shifted to the faster disk as soon as possible during the boot sequence.

Even if you don't autoboot from your hard disk, you can get your Amiga up and running almost at hard disk speed with careful editing of the startup-sequence. The secret is to get the hard disk Mount-ed, Assign stuff to it as quickly as possible, and then call the rest of the commands from the hard disk. A small example:

```
Setpatch                    ; Needed under 1.3
Mount dh0:                  ; crank up the hard disk
dh0:c/Assign c: dh0:c           ; reassign C:
dh0:c/Assign l: dh0:l           ; ditto, L:
dh0:c/Assign libs: dh0:libs     ; ditto, LIBS:
dh0:c/Assign devs: dh0:devs     ; yup
dh0:c/Assign s: dh0:s           ; un-huh
                                ; and so on...
```

Calling the commands with their full pathnames speeds up the execution time a bit. Under AmigaDOS 1.2, further speed can be gained from using the ram: disk or the Addbuffers command. The 1.3 command Resident makes the Assign command faster still: AmigaDOS doesn't have to load it every time the Assign command is used.

After you get the startup-sequence refined to your satisfaction, you'll want to put your software applications into various subdirectories on your hard disk. The trick to this is not to put multiple copies of the WorkBench files from each of the program disks onto your hard disk (as many software manuals direct you to do). Instead, put l directory things into one big l directory; anything that comes out of a libs directory on a disk should go in your hard disk's libs directory; and so on.

Another thing to avoid is copying AmigaDOS 1.2 WorkBench commands onto a hard disk that you want to run under 1.3. If you just copy everything from an applications disk, you might even be mixing 1.1 commands with the stuff on your hard disk. You'll sleep much easier if you pick the version of AmigaDOS you want and run everything under it – especially the parallel.device and the printer.device. If you get 1.3 versions of those mis-matched with earlier versions of the printer drivers, you'll know the first time you try to print something. It won't work.

To install applications on your hard disk, figure out what parts of the application disk's contents are WorkBench files and don't copy those over. If you make a mistake, just copy an original WorkBench 1.x (where x = your choice) *after* you copy all the program disks. With a WorkBench disk in df0:, the CLI command for this is:

```
Copy df0: dh0: all
```

This assumes that you have implemented the standard directories (s, l, libs, devs, devs/printers and so on). If you don't have them, the command above will create them.

The larger the drive or partition, the larger the directory tree it can accommodate. You should put subdirectories in whenever the number of files in a directory becomes too large. You'll *know* when it's too large – the Amiga will seem to take forever to display a directory.

There is no limit to how many levels of subdirectories you can have, and each of them can have a name up to 30 characters long. However, if you keep it simple, you can issue commands from the CLI without interminable typing. Furthermore, only AmigaDOS 1.3 can deal with a command line pathname longer than 80 characters. If you find yourself using a long pathname frequently, you can save yourself some time by using an Assign; for example:

```
Assign ENG: dh0:programming/languagecompilers/English
```

After that, you can get a directory of your sourcecode with the simple CLI command:

```
List ENG:source
```

The Cd command will be much easier to operate if you stick to subdirectory names that are short and sweet, or Assign names to complex directory paths. The current directory option on the some of the 1.3 commands will be useful, too. To copy a file to the directory you're currently in, the command is:

```
Copy Fullpath:filename ""
```

The significant part of that command is the quotes.

Oh say, can you see c:?

Want to add a command to the Amiga's bag of tricks? Simple. Slap it in the c: device and then the program can be called from the CLI or from a script (like the startup-sequence) as readily as List or Type.

There are so many things that are nice to have in the c: device – picture viewers, sound players, music players, screen savers, file checkers, disk checkers, memory checkers, checker checkers – that putting them all in there would slow down system performance on even the fastest hard disk drive.

The solution is a little trick borrowed from the UNIX crowd: make a directory for those other c: things called 'bin' (for 'binary'). A good place for it on the Amiga is as a subdirectory of c:. Put only the native Amiga command programs in c:, and all your other doodads in c:bin. The path to those files is then dh0:c/bin, or c:bin. Then add a Path to the bin subdirectory so that AmigaDOS can find them in the same way it finds c: items. This command, usually best placed in the startup-sequence, is:

```
Path dh0:c/bin Add
```

A benefit of the Amiga's logical devices is that they don't require much typing; you can, for example, get a directory of c: as readily as dh0:c. So assign a device name to the bin directory, too. Then you can perform commands on it without much typing on the command line. The command to do this is:

```
Assign BIN: dh0:c/bin
```

or

```
Assign BIN: c:bin
```

Both of those do the same thing; however, the full path in the first one would permit you to reassign c: as you work, without also reassigning BIN:. If you wanted both to float, the second command would be the one to use.

After you have your BIN: device running, you'll find that the regular c: commands run at maximum speed, and the miscellaneous ones take a moment more for the machine to find them among the large number of entries. This also facilitates changing AmigaDOS versions, since you don't have to deal with anything in the c: device except the native Amiga ones.

Tempting as it may be, beware of adding too many paths. The time required to notify you that a file is not found is greatly affected by how far and wide the computer has to search for it.

MAN: Made

As you were assembling that bin directory, you no doubt noticed that almost all of the programs came with a bit of documentation, generally in a file called something like 'read.me'. However, unless you renamed the read.me files as you went, you overwrote all the read.me files one after the other as you copied them. Even if you did rename them, these files greatly increase the number of entries in the bin directory. To access one of them, you have to wade through the directory and then use Type or More to read them.

The solution to this problem is a single AmigaDOS command, but you have to do a little work first. This trick, too, is borrowed from UNIX. First, you need YAS – Yet Another Subdirectory. You can put it anywhere and call it any name, but for this discussion let's assume you choose 'dh0:manuals'. It's uncreative, but it'll do the job. Put all those little documementation files in the manuals subdirectory. Change their names from Read.ME, Program.DOC, and so on, to something like:

```
ProgramName.MAN
```

where ProgramName is the name of the program for which the file is documentation. You don't really have to have the '.MAN' extension, but if you don't use some kind of naming convention you won't be able to distinguish the executable from the documentation. Sometimes you can't tell from reading the documentation, either, but that's another story.

On most UNIX systems, you can read the 'manual' for a program by calling the UNIX program *man*, which searches its assigned path for a text file having a .man extension and the program name before the dot. So where's the Amiga *man* program? Oh, there are several. However, you don't need it. Amiga DOS 1.3 can handle the task on its own with a single command – after you create an *alias*, that is.

First, though, it would be nice (at least UNIX folks would think kindly of you), if you assigned the logical device MAN: to wherever you put all those documentation files. Then the Alias command is simpler, and takes a few bytes less memory. The Assign command to use is:

```
Assign MAN: dh0:manuals
```

With that in mind, the Alias command to use is:

```
Alias man More MAN:[].man
```

Of course, you should substitute the name of your favorite text reader (like AmigaDOS' Type) for More. Letting AmigaDOS do the work, instead of using a *man* program, does the job in only 40 bytes of memory.

Programming in M2AMIGA

Interfacing with Assembly, Part II

by Anthony Bryant
Copyright © 1989 A. Bryant

This is part of a series of articles on using the Modula 2 programming language, specifically the M2AMIGA Development System from Interface Technologies.

Continuing where he left off, the M2AMIGA programmer comes to grips with the practical side of interfacing assembly language routines...

Disassembling object code

In the course of investigating methods used to pass parameters to assembly procedures, I had to develop a program that would dissect object (.obj) module files (in a special M2AMIGA format, different from the standard Amiga format) to see how this particular M2AMIGA compiler handles the stack frame (or 'activation record').

OM2D (Object M2 Disassembler) is a special purpose program (a re-distributable version is available on this month's *TransAmi* disk in connection with the example programs in this article) which loads and decodes any selected M2AMIGA object file for on-screen presentation and quiet perusal. Each line of disassembly consists of a 32-bit address field, a 16-bit instruction code field plus operand(s), and a mnemonic description of that instruction.

The object code for a module must be relocatable, so all instructions are relocatable and, thus, the address field just points to the location in the object file. Instructions that make address references have to use relative addressing modes, in which the effective address is found by adding a 16 bit offset to the program counter or base address. The exclusive use of this relative addressing mode restricts code size to 32k bytes per object module, however a program may consist of many modules, and is thus not limited.

At the start of an object module is a BRA (branch always) instruction, which jumps to the initialization code (if applicable) for an implementation module, or to the BEGIN in a normal program module. Object code is generated for every procedure in the same sequence as the module source. (A BRA instruction is also generated, directly inline, whenever you use a forward declared procedure.)

Just browsing...

Of course, there is more information contained in an M2AMIGA object file than just the raw code. This includes the externally referenced (XREF) jump table which enables other modules to find the entry points (XDEF) for this module's procedures, storage for string and other constants, and the number of bytes to be reserved for variables, all of which are only resolved later by the linker. For the purpose of this study, or just 'browsing', it was useful to be able just to scroll around in the raw code.

With OM2D, use the cursor keys with repeat action to scroll up or down a line at a time, or use the proportional gadget to scroll a page at a time, or use the mouse to track the scroll. As you scroll, instructions are being disassembled in real time, and displayed with a little assist from the blitter.

All of the mnemonics described (in Part I and II of this article) closely follow the style of the on-screen presentation using OM2D.

Some assembly required

Despite good compiler designs, nothing (yet) surpasses a well-engineered, hand-crafted piece of assembly code for pushing the Amiga to its limits. Of course, there is none of the run-time checking that you get with the M2AMIGA compiler. (Well, it's not needed with a 'well-engineered' piece of assembly code, is it?)

When nothing else will quite do; when you must interface with a set of external (separately assembled and produced) assembly language routines, be prepared to sacrifice something for what you gain.

Do you hear an example? Coming right up!

WarpText (or, beam me up Mr. Kelly)

Bill W. Kelly wrote his *WarpText* routines in order to accomplish direct text transfer to a window's bitmap - one programmer's solution to the problem of speeding up the standard text output. Four stand-alone assembly routines, InitWarpInfo(),

GotoXY(), GetXY() and WarpText(), allow for initializing, outputting text directly to the bitmap, and repositioning by column and row.

Thanks to Bill's selfless distribution of the assembler source and object code for others to study and learn from (see Fish disk #96), I was able to convert *WarpText* to M2AMIGA. Line-by-line conversion of even these short routines was a bit tedious, but the process should be made much easier when m2oc, the object file converter, is released from the M2AMIGA people. (I couldn't wait.)

Presented here are the definition and implementation module interfaces for *WarpText*, organized as an M2AMIGA library module. *WarpText.def* defines the requisite WarpInfo structure as a record and sets out the four procedure definitions, while *WarpText.imp* contains the original relocatable assembly code. (See *WarpText.def* and *WarpText.imp* at the end of this article.)

Parameters passed (and present)

With the procedures InitWarpInfo(), GotoXY() and WarpText(), parameters passed via registers interface directly with the assembly code. These well-behaved routines preserve all registers they intend to use and restore them when finished (the MOVEM instruction is a very efficient way to stack and unstack a large number of registers). Workspace for temporary variables is built directly into the WarpInfo structure, so we do not need access to global variables via register A4 or local variables via register A5, and can dispense with the extra entry and exit code.

Procedure GetXY() needs some 'glue' code, however, in order to return its register values (in D0 and D1) to the call-by-reference VAR x, y. (Recall: Only D0, D1, A0 and A1 are free for scratch use.) The term 'glue' normally refers to the assembly code that is used to bind with the calling-sequence for a high-level language. Here, though, it is the other way around; 'glue' means the high-level code needed to stick to assembly!

Because of the need to access the stack, a normal *activation record* is established making register A5 accessible. (Register A4 is also made accessible, but we won't be using it.)

You cannot mix register and stack parameter-passing techniques in the same procedure definition, so SETREG() is used to move the first parameter (wInfo) from the stack and into address register A0, where it needs to be. The GetXY() assembly code is then called, with INLINE(). With call-by-reference VAR-type parameters, it's the address of the variable that is pushed on the stack, so the REG() function is used to copy each register value (in D0 or D1) into the actual variable location.

Once the transfer to variables is completed, the parameters - all addresses - on the stack are no longer needed. The compiler does the work of restoring the stack frame in the exit code. The extra entry and exit code generated by the compiler is shown for the sake of completeness.

Alternately, of course, I could have split up the original GetXY() into two separate functions, one each to return the column and row (in D0), but this way is shorter, has the benefit of demonstrating the use of SETREG() and REG() and also remains true to the original code.

The WarpText() code is an interesting example of bit-twiddling a TextFont structure in order to adjust to different-sized fonts, as well as of the fast bitmapping technique. As an aside, the actual mapping of the bits takes place in the code at label *MoveChar*. Note that different style algorithms (underlined, bold, italic, etc.) are *not* supported here (sacrificed, no doubt, for the sake of speed) but it does open the door for the testing of one's own algorithms! In fact, the whole technique of intermixing high-level language with low-level can be used as a staging area from which to check out short, but interesting bits of assembler code.

Writing directly to the bitmap also circumvents window clipping (actually circumventing layers library routines). The *WarpText* routines were intended for printing to a custom screen, and write into bitplane memory directly, so use with caution: don't display text that would extend beyond the edges of your display area.

WarpText() interprets a linefeed character (as well as carriage return and formfeed) that is embedded within the text. You can also use GotoXY() to re-position the text directly. Character values outside of the font's *loChar* and *hiChar* limits are replaced with the 'empty-box' character, hex $80 in the topaz font. (In a later version of *WarpText*, frustrated at not finding the 'empty-box' character, Bill made an 'empty-box' bitmap character from scratch, so as not to be dependent on any particular font.)

Using the supplied version of OM2D on this disk, open the file requester (which is pre-set to the obj directory) and select 'WarpText.obj'. A quick browse through 'WarpText.obj' reveals the inline code laid out in the same order as in the implementation module.

InitWarpInfo()	at address 0000017A
GotoXY()	at address 000001F4
GetXY()	at address 00000206
WarpText()	at address 0000023E

Warp speed

The program *wtdemo* is a quick (don't blink or you'll miss it) test of these routines - see *wtdemo.mod* at the end of this article. It simply allocates a global WarpInfo structure, and then uses LoadFont() to open the standard topaz size 8 ROM font. To select a font from disk, change the font name, size and flags (flags should be 'diskFont') here.

Because LoadFont() opens either a ROM or disk font, depending on what flags you give it, it needs to access the 'diskfont.library' in order to call OpenDiskFont(). If M2AMIGA can't

find the 'diskfont.library' on your SYS: disk, the program just quits in disgust.

Note that the TextAttr structure, like all local variables on the stack, is only temporary. Once you have opened a font, TextAttr is no longer needed and, once that procedure has ended, no longer available; information about the font is now to be found in the TextFont structure.

The use of the Arts module's Assert() procedure provides a clean way to exit in case we encounter any problems along the way. The first parameter is a boolean value which, if FALSE, causes an error requester to appear, using the second parameter as a text message to be displayed; if the Assert() condition is TRUE, the program just continues.

The rest of the initialization is handled by InitWarpInfo(). No need to SetFont(), since InitWarpInfo() also sets up the font data. Only non-proportional fonts are supported, and be sure not to pass any NIL pointers to any WarpText routines, since the assembly code does no checking for such exceptions.

Both methods of repositioning text, that is, embedding a linefeed char (12C) in the text, and using GotoXY() to offset from the last character position, column and row, (found by using GetXY()) are demonstrated. Notice that when finished with the font that we loaded, we use the CloseFont() routine to mark the font as not in use, so that at some later time (when the system needs free memory) it will get unloaded. While not strictly necessary with ROM fonts (you can't unload them) it is a good idea to get into the habit of freeing resources.

I found WarpText an interesting and impressive lot of coding. Thanks to Bill W. Kelly for sharing his contribution with the Amiga community.

The M2AMIGA compiler especially simplifies the job of interfacing with internal Amiga library routines and external (separately assembled) roll-your-own routines. Library and inline code is generated and handled very similar to the way an Assembler would do it. This makes for very efficient and compact code.

In practice, only short bits of such hand-optimized code should ever need conversion interfaces to M2, otherwise you would be better off programming with an Assembler in the first place. Ah, but I would miss the high-level language environment, with its automatic handling of code, data and stack space. Perhaps if the M2AMIGA Development System had a built in Assembler...

* * *

M2AMIGA Development System is available from Interface Technologies. The WarpText code and WarpText is Copyright 1987 by Bill W. Kelly. OM2D (Object M2 Disassembler) is Copyright 1988 by A. Bryant

Listing 1: ***WarpText.def***

```
DEFINITION MODULE WarpText;     (* WarpText 1.0 *)

(*
 * "The WarpText code, and the name 'WarpText', is
 *    Copyright 1987 by Bill W. Kelly"
 *
 *  M2Amiga adaptation by Anthony Bryant
 *)

FROM SYSTEM IMPORT
 ADDRESS;

FROM Graphics IMPORT
 BitMapPtr,TextFontPtr;

TYPE
   WarpInfo = RECORD
    textFont: TextFontPtr;    (* Pointer to a TextFont structure to use *)
    bitMap: BitMapPtr;        (* Pointer to a BitMap structure to use *)
    whichPlane: CARDINAL;     (* Which bitplane to render into *)
    left: CARDINAL;           (* Left edge of 'window' in char loc's *)
    top: CARDINAL;            (* Top edge of 'window' in char loc's  *)
    width: CARDINAL;          (* Width of 'window' in char loc's  *)
    height: CARDINAL;         (* Height of 'window' in char loc's *)
    windowTop: ADDRESS;       (* <internal> Address of top of 'window' *)
    curLine: ADDRESS;         (* <internal> Address of start of current line *)
    lastLine: ADDRESS;        (* <internal> Address of start of last line  *)
    curX: CARDINAL;           (* <internal> Current X position  *)
    lastX: CARDINAL;          (* <internal> Maximum X position on a line  *)
    bPMod: CARDINAL;          (* <internal> Max possible chars on a line in bp *)
    modulo: CARDINAL;         (* <internal> Add this to get to next line  *)
   END;
   WarpInfoPtr = POINTER TO WarpInfo;


(*  InitWarpInfo  -    initializes the WarpInfo structure for use
 *)
PROCEDURE InitWarpInfo(wInfo{8}: WarpInfoPtr);


(*  GotoXY  -    sets cursor to given character positions, x column, y row
 *)
PROCEDURE GotoXY(wInfo{8}: WarpInfoPtr; x{0}: CARDINAL; y{1}: CARDINAL);


(*  GetXY  -    returns current cursor character positions, x column, y row
 *)
PROCEDURE GetXY(wInfo: WarpInfoPtr; VAR x, y: CARDINAL);


(*  WarpText  -     emits text at approx 13,500 characters per second
 *)
PROCEDURE WarpText(wInfo{8}: WarpInfoPtr; text{9}: ADDRESS; count{0}: LONGCARD);


END WarpText.def
```

Listing 2: ***WarpText.imp***

```
DEFINITION MODULE WarpText;     (* WarpText 1.0 *)

(*
 * "The WarpText code, and the name 'WarpText', is
 *    Copyright 1987 by Bill W. Kelly"
 *
 *  M2Amiga adaptation by Anthony Bryant
 *)

FROM SYSTEM IMPORT
 ADDRESS;

FROM Graphics IMPORT
 BitMapPtr,TextFontPtr;

TYPE
   WarpInfo = RECORD
    textFont: TextFontPtr;    (* Pointer to a TextFont structure to use *)
```

```
    bitMap: BitMapPtr;        (* Pointer to a BitMap structure to use *)
    whichPlane: CARDINAL;     (* Which bitplane to render into *)
    left: CARDINAL;           (* Left edge of 'window' in char loc's *)
    top: CARDINAL;            (* Top edge of 'window' in char loc's  *)
    width: CARDINAL;          (* Width of 'window' in char loc's  *)
    height: CARDINAL;         (* Height of 'window' in char loc's *)
    windowTop: ADDRESS;       (* <internal> Address of top of 'window' *)
    curLine: ADDRESS;         (* <internal> Address of start of current line *)
    lastLine: ADDRESS;        (* <internal> Address of start of last line  *)
    curX: CARDINAL;           (* <internal> Current X position  *)
    lastX: CARDINAL;          (* <internal> Maximum X position on a line  *)
    bPMod: CARDINAL;          (* <internal> Max possible chars on a line in bp *)
    modulo: CARDINAL;         (* <internal> Add this to get to next line  *)
   END;
   WarpInfoPtr = POINTER TO WarpInfo;


(*  InitWarpInfo  -    initializes the WarpInfo structure for use
 *)
PROCEDURE InitWarpInfo(wInfo{8}: WarpInfoPtr);


(*  GotoXY  -    sets cursor to given character positions, x column, y row
 *)
PROCEDURE GotoXY(wInfo{8}: WarpInfoPtr; x{0}: CARDINAL; y{1}: CARDINAL);


(*  GetXY  -    returns current cursor character positions, x column, y row
 *)
PROCEDURE GetXY(wInfo: WarpInfoPtr; VAR x, y: CARDINAL);


(*  WarpText  -    emits text at approx 13,500 characters per second
 *)
PROCEDURE WarpText(wInfo{8}: WarpInfoPtr; text{9}: ADDRESS; count{0}: LONGCARD);


END WarpText.def
```

Listing 3: *WTdemo.mod*

```
MODULE wtdemo;   (* \/\/arpText Quickie Demo *)

(* M2Amiga demo program written by Anthony Bryant *)

FROM SYSTEM IMPORT
 ADDRESS,ADR,BYTE,CAST;

FROM Arts IMPORT
 Assert, TermProcedure;

FROM DiskFont IMPORT
 OpenDiskFont;

FROM Exec IMPORT
 MemReqs, MemReqSet, AllocMem, FreeMem, GetMsg, ReplyMsg, WaitPort;

FROM Graphics IMPORT
 BitMap, BitMapPtr, FontStyles, FontStyleSet, FontFlags, FontFlagSet,
 TextAttr, TextAttrPtr, TextFont, TextFontPtr,
 CloseFont, OpenFont;

FROM Intuition IMPORT
 IntuiMessage, IntuiMessagePtr, IDCMPFlags, IDCMPFlagSet, NewWindow,
 WindowFlags, WindowFlagSet, ScreenFlags, ScreenFlagSet, Window, WindowPtr,
 CloseWindow, OpenWindow;

FROM WarpText IMPORT
 InitWarpInfo, GotoXY, GetXY, WarpText, WarpInfo, WarpInfoPtr;


(* OpenNewWindow - prepare a standard window to hold the text
 *)
PROCEDURE OpenNewWindow(): WindowPtr;
 VAR
   wdef: NewWindow; (* temporary structure *)
 BEGIN
  WITH wdef DO
   leftEdge:= 0; topEdge:= 0; width:= 480; height:= 200;
   detailPen:= 0; blockPen:= 1;
   idcmpFlags:= IDCMPFlagSet{closeWindow};
   flags:= WindowFlagSet{windowDrag, windowDepth, windowClose,
                         activate, noCareRefresh};
   firstGadget:= NIL;
   title:= ADR("\/\/arpText Quickie Demo");
   screen:= NIL; bitMap:= NIL;
   type:= ScreenFlagSet{wbenchScreen};
  END;
  RETURN OpenWindow(wdef);
 END OpenNewWindow;


VAR
   wp: WindowPtr;
   wi: WarpInfoPtr;

PROCEDURE Cleanup();  (* this will be our exit routine *)
 BEGIN
  IF (wi # NIL) THEN  FreeMem(wi, SIZE(WarpInfo)); END;
  IF (wp # NIL) THEN  CloseWindow(wp); END;
 END Cleanup;


(* strlenConvert -  returns the length of a null-terminated string AND
 *              converts a line end char,"\", to an embedded LF char, 12C
 *)
PROCEDURE strlenConvert(string: ADDRESS): LONGCARD;
 VAR
  len: LONGCARD; txt: ADDRESS;
 BEGIN
  len:= 0; txt:= string;
  WHILE (CAST(CHAR,txt^) # 0C) DO INC(len); INC(txt); END; (* find length *)
  DEC(txt);
   (* if last char is special line end char, "\" then replace with LF char *)
  IF (CAST(CHAR,txt^) = "\") THEN txt^:= CAST(BYTE,12C); END;
  RETURN(len);
 END strlenConvert;


(* LoadFont - open specified font in ROM or from disk, return pointer to font
 *)
PROCEDURE LoadFont(fontName: ADDRESS; fontSize: CARDINAL;
                   fontType: FontFlagSet): TextFontPtr;
 VAR
   ta: TextAttr; (* a temporary structure *)
 BEGIN
  WITH ta DO
   name:= fontName;
   ySize:= fontSize;
   style:= FontStyleSet{};  (* normalFont - NO algorithms available *)
   flags:= fontType;
  END;
  IF (romFont IN fontType) THEN
    RETURN OpenFont(ADR(ta));
  ELSIF (diskFont IN fontType) THEN
    RETURN OpenDiskFont(ADR(ta));
  ELSE
    RETURN NIL;
  END;
 END LoadFont;


CONST
  maxlines = 19;
VAR
  text: ARRAY [0..maxlines-1] OF ADDRESS;
  length: ARRAY [0..maxlines-1] OF LONGCARD;
  msg: IntuiMessagePtr;
  class: IDCMPFlagSet;
  i,column,row: CARDINAL;
```

```
BEGIN  (* and away we go! *)
  TermProcedure(Cleanup);  (* leave a tidy Amiga *)

  wp:= WindowPtr(OpenNewWindow());
  Assert(wp # NIL, ADR("Can't open window!"));  (* oops, exit with error *)
  wi:= WarpInfoPtr(AllocMem(SIZE(WarpInfo), MemReqSet{public}));
  Assert(wi # NIL, ADR("Can't allocate memory!"));  (* oops, again and exit *)

  (* O.K. so far, now let's fill the text array with some lines of text
   * Note that we are using a "\" to represent a linefeed at line end
   *)
  text[0]:= ADR("Thanks to Bill W. Kelly for  \/\/arpText (1.0)  routines.\");
  text[1]:= ADR("As you just saw, this was a quickie test drive with them.\");
  text[2]:= ADR("I have created DEFINITION and IMPLEMENTATION MODULEs as a\");
  text[3]:= ADR("Modula-2 interface, to use like any other library module.\");
  text[4]:= ADR("M2Amiga programmers can now access FAST text algorithms, \");
  text[5]:= ADR("      written in 100% assembly language code!\");
  text[6]:= ADR(" \");
  text[7]:= ADR("You should be able to attain a speed of 13,000 chars/sec.\");
  text[8]:= ADR("These routines were designed for speed, so use with care.\");
  text[9]:= ADR("Specifically, these routines do NO clipping in the window\");
 text[10]:= ADR("    so YOU will have to manage your own clipping!\");
 text[11]:= ADR("Also, make certain that you have properly initialized the\");
 text[12]:= ADR("      WarpInfo structure   (see WarpText.def)\");
 text[13]:= ADR(" \");
 text[14]:= ADR("Choose:  GotoXY (specific column and row) or else embed a\");
 text[15]:= ADR("CR, LF or FF control character in order to redirect text.\");
 text[16]:= ADR("A TextFont's loChar and hiChar bounds are checked and if \");
 text[17]:= ADR("out-of-bounds chars are replaced with the empty-box .....\");

 text[18]:= ADR("\/\/arp\/\/arp\/\/arp\/\/arp"); (* no line end char here *)

  (* now we need to fill the length array AND replace (embed) the
   * linefeed character - "\"  at line end with the real thing.
   *)
  FOR i:= 0 TO maxlines-1 DO
    length[i]:= strlenConvert(text[i]);
  END;

  (* now set up important WarpInfo parameters !*)
  WITH wi^ DO
    textFont:= TextFontPtr(
              LoadFont(ADR("topaz.font"), 8, FontFlagSet{romFont}) );
    bitMap:= BitMapPtr(wp^.rPort^.bitMap);
    whichPlane:= 0; (* 0 or 1 ONLY! *)
    left:= 1;  top:= 2;       (* left, top, width & height are *)
    width:= 58; height:= 21;  (* in character columns and rows *)
  END;
  Assert(wi^.textFont # NIL, ADR("Can't open your Font!")); (* oops ! *)

  InitWarpInfo(wi); (* other parameters are initialized by InitWarpInfo() *)

  (* ahead, \/\/arp speed! *)
  FOR i:= 0 TO maxlines-2 DO
      WarpText(wi, text[i], length[i]);
  END;

  GotoXY(wi, 0, 19);  (* re-position the "invisible cursor" *)
  WarpText(wi, text[18], length[18]);  (* print this short piece *)
  GetXY(wi, column, row); (* get last position into VARs column & row *)
  GotoXY(wi, column+1, row);   (* advance to next column *)
  WarpText(wi, text[18], length[18]);  (* print that last piece again *)

  GotoXY(wi, 0, 21);  WarpText(wi, ADR(" Cheers!"), 8);

  CloseFont(wi^.textFont);  (* finished with font *)

  LOOP   (* now sit and wait for closeWindow message *)
     WaitPort(wp^.userPort);
     msg:= IntuiMessagePtr(GetMsg(wp^.userPort));
     IF (msg # NIL) THEN
       class:= msg^.class;
       ReplyMsg(ADDRESS(msg));
       IF (closeWindow IN class) THEN EXIT; END;
     END;
  END; (* LOOP *)
  (* TermProcedure() kicks in here *)

END wtdemo.mod
```


3D OPTIONS

CONVERT IFF BITMAPPED PICTURES TO 2D AND 3D OBJECTS!!!

Now You Can Make Use of All Those
DPaint and Digitized Pictures in
Your Favorite 3D or CAD Package

VIDEOSCAPE 3D, AEGIS DRAW, INTROCAD, MCAD, PROFESSIONAL PAGE, POSTSCRIPT DXF (AUTOCAD...) & More.

TO ORDER CALL
1 800-628-2828 Ext 829

We accept VISA, MASTER CARD and AMERICAN EXPRESS

Only $49.95

Or Send Check or Money Order to:
Rainbows Edge Productions
4412 4th Avenue Suite 2
Brooklyn, NY 11220

All product names are Trademarks of their respective companies

SAFE HARBOR SOFTWARE & PERIPHERALS

Great Prices! Shipping based on weight and zone.
For Information & Catalog Call:
Voice 414-544-2066
Pursuitable BBS 414-544-6567

Software

Ami Alignment Kit	$ 36.00
ANIMagic	$ 59.99
Atalk III	$ 64.69
ARexx	$ 31.10
B.A.D	$ 33.00
Battlehawks	$ 33.89
Blood Money	$ 27.99
Broadcast Titler	$195.00
CLIMate	$ 27.73
CygnusEd	$ 64.69
Deluxe Paint III	$105.00
Demon's Winter	$ 29.97
Devpac Amiga	$ 64.69
Digi Paint 3	$ 75.00
Digi View Gold	$143.10
GOMF 3.0	$ 27.28
Gunship	$ 37.99
Hi Soft BASIC Professional	$115.00
Inovatools	$ 52.37
Interfont	$ 76.00
Lattice C 5.0	$190.00
Level II (Dr. T)	$219.00
Microfiche Filer +	$111.00
Movie Setter	$ 60.00
PageStream	$131.00
Pen Pal	Call
Performer (Elan)	$ 41.00
Power Windows 2.5	$ 59.09
Professional Draw	$130.00
Prof. Page Templates	$ 40.00
Project D	$ 32.00
Quarterback	$ 46.21
SuperBase Personal 2	$ 90.00
SuperBase Professional 3.0	$200.00
SuperPlan	$ 99.97
Turbo Silver	$120.00
Video Effects 3D	$128.00
Who! What! When! Where!	$ 59.99
WShell	$ 31.10

Abacus Books/Sony Diskettes

Amiga C for Beginners	$ 15.00
Amiga DOS Inside/Out	$ 16.30
Amiga Machine Language	$ 16.30
Amiga System Programmer	$ 26.25
DS/DD Sony 3.5" Disks (50)	$ 72.00
ROM Kernal Ref. Manual	$ 27.50

Hardware

501 Memory Clones	$170.00
68030 GVP 25MHz	Call
8-Up! 0K	Call
Amigen	$145.00
Dual Serial Port ASDG	$249.00
Floppy Drive 3.5" (AIR) ext.	$169.95
Floppy Drive, Silent	$150.00
Floppy Internal 2000	$120.00
HardFrame 2000	Call
Live! 2000	$315.00
Midget Racer CSA	$350.00
Midi Gold	$ 67.00
Mini Megs 512K	$325.00
Modem, Supra 2400 Int.	$145.00
Spirit Brd 500/1000 0K	$219.00
Supra SCSI Cntrlr 500/1000	$220.00
Supra SCSI Cntrlr 2000	$210.00
Tool Box	$215.00

Orders Only Please:
Outside WI: **800-544-6599**
Inside WI: **414-544-6599**
Visa/MC/CODs

2414 Pendleton Place ■ Waukesha, WI 53188 ■ 9 AM to 5 PM M-F

Toy - A Small Compiler for the Amiga

Part 1 of a series on compiler construction

by Chris Gray

To most people, compilers are black boxes. They read an input text file containing source and magically write out an object file containing equivalent machine instructions. In actuality, there is nothing magical about a compiler. A compiler doesn't even have to be very complex or very large. Since programmers are quite demanding, however, the compiler writer (being a demanding programmer), is not likely to be satisfied with a compiler that is less than perfect. Thus, the typical compiler soons ceases to be simple and small. Such minor details as complex data types, good code and good error recovery can add thousands of lines to a compiler.

Such an embellished compiler is probably not fully understood even by its author, let alone anyone else. When people have requested the source to my Draco compiler, so they can learn from it and add some stuff they want, I have been somewhat reticent. To be honest, it is a mess inside. This situation has led me to write a new compiler, one designed to be simple and understandable, but nevertheless a real compiler for the Amiga.

The result of this is the Toy compiler, which I will describe in this series of articles. It is quite simple in operation, and uses the same techniques that the Draco compiler itself uses. It is written in Draco, which is freely available, so people can tinker with it without having to buy a commercial compiler. In spite of its simplicity, it is a true compiler. It goes from a source file to an executable object file containing MC68000 instructions. It does not use an interpreter, or threaded code, or any other such technique for execution – it emits straightforward (and not overly efficient) instructions to accomplish the steps specified by the input source program statements.

In this first article of the series, I will describe the language that is being compiled, giving some example programs written in Toy. I will also outline the general structure of compilers, and the Toy compiler in particular. Techniques used in commercial and experimental compilers will be mentioned, as will many of the problems that compiler writers must contend with, and the decisions they must make.

The Toy language

One of the first decisions to be made when designing a compiler is that of deciding what language is to be compiled. To keep the compiler simple, we want a simple language. This pretty well rules out a complete version of an existing language. Most Amiga programmers would prefer a C-like language, since that is what they use. I don't feel that is a good choice for a number of reasons. C is normally used with a preprocessor (either a separate program or built into the compiler), and having one would increase the complexity of the project. C's syntax is ambiguous (for example, **fred();** is either a declaration or a function call, depending on where you see it), thus complicating any error recovery. C's inside-out declarations are a pain to handle (it took me a couple of tries in the C compiler I did at work).

Lastly, C's automatic declarations and unusual ideas about scope (anything declared as **extern** is global, regardless of where you declare it) add unneeded hazards. I didn't see any gain in inventing a whole new language, however, so I went with what I am most familiar with – Draco. The Toy language is a proper subset of Draco. All of the Toy programs I have written compile and run just fine under the Draco compiler.

For those not familiar with Draco, I will describe the Toy language here, using a simple grammar and some examples. A Toy program consists of any number of procedures, which may or may not return a result. Each procedure can have named, typed parameters. Variables global to the file and local

to procedures exist, and have similar types. Executable statements include the assignment statement, the **if** statement, the **while** statement, and some simple input/output statements. Operators usable in expressions include +, -, * and /. Available in conditions are the standard comparison operators. A grammar for the language is presented in Figure 1.

Many parts of the Toy compiler's parser will follow this grammar quite closely. Note in particular that the definition of **<expression>** eventually cycles around to mentioning **<expression>** on the right-hand side. This recursive definition is echoed in the recursive descent parser used in Toy.

The compiler places many limitations on acceptable programs that are not described by the grammar. Types of grammars exist that can contain all of the semantic information needed, but building a correct one of those is more work than building a compiler, so I have not done so. Those interested in such things should read the Algol68 report. (Actually, I'm being sarcastic here – the report is, in my opinion, unreadable without a great deal of study.) There are two main types of limitations. The first type are due to the specific implementation of the compiler. For example, each **if** statement is limited, by an array in the parsing code, to ten **elif** parts. This could be handled by using a dynamically-allocated linked list, but is not a

```
<program>                ::= <variables>* <proc>*
<variables>              ::= <type> <id-list> ';'
<empty>                  ::=
<type>                   ::= 'int' | 'bool'
<id-list>                ::= <id> | <id-list> ',' <id>
<id>                     ::= Any sequence of letters, digits and underscores, starting with a letter or an underscore
<proc>                   ::= 'proc' <id> '(' <parameters> ')' <result-type> ':' <variables>* <statement-list>
                                 <proc-result> 'corp' ';'
<parameters>             ::= <empty> | <par-seqs>
<par-seqs>               ::= <par-seq> | <par-seqs> ';' <par-seq>
<par-seq>                ::= 'int' <id-list>
<result-type>            ::= <type> | 'void'
<proc-result>            ::= <empty> | <expression>
<statement-list>         ::= <statement> | <statement-list> ';' <statement>
<statement>              ::= <if-statement> | <while-statement> | <readln-statement> | <write-statement>
                                 | <assignment-statement> | <empty>
<if-statement>           ::= 'if' <expression> 'then' <statement-list> <elif>* <else> 'fi'
<elif>                   ::= 'elif' <expression> 'then' <statement-list>
<else>                   ::= <empty> | 'else' <statement-list>
<while-statement>        ::= 'while' <expression> 'do' <statement-list> 'od'
<readln-statement>       ::= 'readln' '(' <id-list> ')'
<write-statement>        ::= 'write' '(' <write-list> ')'
<write-list>             ::= <write-unit> | <write-list> ',' <write-unit>
<write-unit>             ::= <string> | <expression>
<assignment-statement>::= <id> ':=' <expression>
<expression>             ::= <comparison>
<comparison>             ::= <add-sub> | <add-sub> <comp-op> <add-sub>
<comp-op>                ::= '=' | '~=' | '<' | '<=' | '>' | '>='
<add-sub>                ::= <mul-div> | <mul-div> <add-sub-op> <mul-div>
<add-sub-op>             ::= '+' | '-'
<mul-div>                ::= <unary> | <unary> <mul-div-op> <unary>
<mul-div-op>             ::= '*' | '/'
<unary>                  ::= <unit> | '-' <unary> | '+' <unary>
<unit>                   ::= <id> | <proc-call> | <number> | '(' <expression> ')' | 'true' | 'false'
<proc-call>              ::= <id> '(' <expr-list> ')'
<expr-list>              ::= <expression> | <expr-list> ',' <expression>
<number>                 ::= Any sequence of decimal digits
```

Figure 1: A grammar for Toy

The **::=** symbol means that the term on the left can be any of the |-separated sequences on the right. Items in apostrophes must appear exactly as given. Items in angle brackets (< and >) must be replaced by one of their right-hand sides in order to make an acceptable program. A following * means that the term can appear any number of times (including none).

problem in a toy compiler. The other type of limitation imposed by the compiler is that of semantic correctness: the compiler will reject things that don't make sense, even if they are correct according to the grammar. For example:

```
bool flag;

flag := 6;
```

is flagged as an error even though it fits the grammar just fine. This is because Toy is a strongly-typed language. The compiler also checks such things as parameter counts and types on procedure calls.

The alert reader will already have noticed that Toy isn't a very impressive language. You cannot write the world's best Amiga program using it. It is pretty difficult to write *anything* useful in it. It has only two data types: **int** and **bool**. It has no arrays, no structures, no pointers, no named constants, and a pretty wimpy I/O system. Rectifying any of these deficiencies would have significantly increased the size and complexity of the Toy compiler, making it less suitable as a tutorial compiler. Interested programmers can add many features to Toy themselves. I will be giving hints on how to go about this as I discuss the various portions of the compiler.

In the following discussions of the Toy language and compiler, I will assume that the reader is familiar with at least one programming language, and will be able to understand the general outline of what the various programs are doing without me explaining what all the various language constructs mean. A detailed understanding is not necessary to get the general outline of how the compiler works.

Sample programs

A tradition I've held on to for my compilers is that the first program I try to write is one that prints out "Hello there world!". Here is the Toy version:

```
proc main()void:

   write("Hello there world!\n");
corp;
```

As the grammar indicates, a Toy program can have a series of procedure bodies. This program has one procedure, called *main*, which has no parameters (there is nothing between the parentheses after the procedure name), and returns no result; the special result-word **void** means that no result is returned. **write**, in this case, causes the given string to be written out. The pair of characters **\n** represents the newline character, and indicates that the output line should be ended at that point. **corp** is simply **proc** spelled backwards: program constructs in Toy, as in Draco, are usually ended by a backwards word that matches a beginning word for the construct. This style is strongly disliked by several vocal people. I once felt that way myself, but changed my mind after using it for a while. It makes it quite clear what construct is being closed, something that C's braces or Pascal's **BEGIN-END** pairs fail to do.

My second program is traditionally a recursive solution to the 'Towers of Hanoi' problem. In the unlikely event that you haven't heard of it, I'll describe it briefly. The problem set-up consists of three vertical pegs, on which are placed several round disks. The disks all start on the first peg, and must be moved to the third peg, with the second peg available for intermediate operations. The disks are of varying sizes, and start out in order, with the largest on the bottom, and the smallest on the top. They must end up the same way on the third peg. The only valid move is to move a single disk from one peg to another. At no time may a disk rest on another which is smaller. The story says that there are some priests on a mountaintop who started doing the problem when the world was created, and when they are done, the world will end. It turns out that the traditional 64 disks would take so long to move according to the rules that we need have no fear of the world ending.

The standard solution is a recursive one consisting of three steps, dealing with *N* disks:

1) Recursively using this method, move the top *N - 1* disks from the starting peg to the spare peg.

2) Move the *N*'th disk from its starting peg to its target peg.

3) Recursively using this method, move the top *N - 1* disks from the spare peg to the target peg.

There is a special case - if *N* is zero, don't do anything. Those not familiar with recursion may want to try this out using stacks of small pieces of paper. Remember that the meanings of 'start peg', 'spare peg' and 'target peg' change as you recurse down.

Listing 1 at the end of this article shows **hanoi.t**, the Toy implementation of this solution to the puzzle. I don't recommend that you try it with 64 disks - your Amiga will turn to dust long before it is done! Four or five disks makes a nice test. This program is already showing some weaknesses in Toy - a separate routine 'printPeg' is used to print out the name of a peg. Proper languages would allow string variables, which could be printed out directly. Procedure *hanoi* implements the above solution. Moving a disk consists of printing out a message saying where to move it. Procedure *main* (all Toy programs must have a procedure called *main* - this is where execution starts) asks for and reads in the count of disks to be used, then does the top level call to *hanoi*.

Listing 2 shows **shoot.t**, a program that actually manages to implement a little game in Toy. It uses the cursor and colour control of the Amiga's console device to provide very primitive graphics and animation. The program draws a simple scene consisting of the ground, a 'gun' (a square box sitting on the ground) and a 'target' (a differently-coloured diamond shape, also sitting on the ground). Shells are shot from the gun

at a forty-five degree angle at a speed specified by the player. Their course is shown on the screen and the game is over when one hits the target dead on. The game maintains a score, which is the number of shells shot, and maintains a message area for things like error comments.

shoot is sufficiently complicated that I added some comments. For the non-C and non-Draco programmers out there, comments in Toy (not mentioned at all in the above grammar!) consist of any text between an opening /* and a closing */. The program starts off by declaring a bunch of global variables. I have capitalized their names to make them look like constants, but the Toy compiler takes no notice of that and would be quite happy to let me change their values. They are set up once in the main program, then not changed throughout the game. Following some global variables, the code starts. *random* is the first Toy procedure we've seen that returns a result. The result type of the procedure is specified as **int**, by the word **int** after the parameters in the procedure header. The actual value returned is given by the expression before the **corp**. This point also is not mentioned by the grammar. The seven procedures after *random* send escape sequences to the console window to accomplish various actions, such as moving the cursor, specifying boldface or normal type, and changing the foreground colour of the text. Consult the Amiga technical manuals for details on these escape sequences.

Next are several routines that use these special control routines to do higher-level game-related actions, such as drawing the ground, gun and target. *hitTarget* is one of the largest procedures in the program: it plots the course of the thrown shell, drawing it as an asterisk, and checking to see if it goes off the edge of the screen, hits the target, or just hits the ground. A "-BOOOM" is left wherever the shell hits the ground. Note the final **if** statement in the routine: in a more complete language, the condition would have used a logical **and** operation, rather than an extra **if** statement.

The final routine, *main*, is, as usual, where execution starts. Since Toy has no way to pre-declare procedures, *main* must be the last procedure in the source file, since every other procedure is either called by *main* or by something that *main* calls, etc. Note that this restriction means that mutually recursive procedures cannot be used in Toy. The program requests the screen size so that it knows how to do the cursor addressing for correct positioning. The size should be 23 for a normal non-interlaced screen, and 48 for an interlaced one. The console device can tell the program how big the window is, but since Toy can only read integer values, it would not be possible to understand the string that the console device returns.

General outline of the Toy compiler

The Toy compiler is completely stand-alone. It does not need any other programs or libraries to do its job. In particular, it does not make use of any compiler-construction tools, such as lexer generators or parser generators. The parser generator *Yacc* (Yet Another Compiler-Compiler) has been discussed by Eric Giguère in his series of articles in earlier issues of *Transactor for the Amiga*. *Yacc* and its relatives are often used on UNIX™ systems to generate parsers of various kinds. Along with *Yacc*, UNIX offers *Lex*, a program to generate lexical scanners. These, and other similar tools, can simplify some aspects of compiler writing, but I prefer the flexibility of custom code, so have not used any for either Draco or Toy.

Like any other program that reads input, writes output, parses command lines and prints error messages, a compiler requires an interface to the operating system. In the case of a compiler, the interface is made more complex by the strict structure required of the output machine-code file. In the case of AmigaDos, the structure of an executable file is described in the *AmigaDos Technical Reference Manual*. Note that there are two variants of an object file. The first variant, normally produced by compilers and assemblers, is an unlinked one. This form is not directly executable by the operating system since it makes references to support and library routines that are provided by the compiler libraries and/or operating system. Such a file is linked with similar files containing libraries and interfaces to the operating system to produce a final, fully linked object file. This latter type of file *is* directly executable by the operating system. I chose to keep Toy as simple and complete as possible, so it directly produces the final linked version. This means that the user does not need access to a linker (such as *ALink* or *BLink*) in order to compile and run Toy programs. This implies that the Toy compiler itself must supply any support routines that are needed. For Toy, these consist of the ability to read and write integers and to write strings. The two Toy source files dealing with the system interface and generation of the run-time routines are the only ones that are system-dependent. The remainder of the compiler does not care where it is running, or where its output is going – it is only concerned with the input source and the generated MC68000 instructions.

The system interface supplies input data in a raw form – as a sequence of ASCII characters. The first step in compilation is to break this sequence up into larger, more meaningful pieces. These pieces are such things as identifiers, reserved words such as **proc** and **if**, numbers, punctuation marks and string constants. This process is called 'lexical analysis', 'lexical scanning' or 'tokenization'. The output from this phase of a compiler is a sequence of 'tokens' that represents the input program at a higher level than that of a sequence of characters. The tokenizer is also responsible for skipping over comments, blank lines, spaces, etc. Many compilers do not actually keep the tokenized form of the program around – they simply call a routine to get the next token when it is needed, and then discard it when they go on to the one after that. Toy uses this technique. There are not many error messages associated with tokenization – about the only thing that can be complained about are characters not in the set of characters that the language uses.

For a C compiler, the sequence of events described here is somewhat different. First, the input text is very roughly tok-

enized – just enough for the preprocessor to spot its directives. The resulting 'preprocessor tokens' are handled by the preprocessor to do macro replacement and conditional compilation. The output from the preprocessor is another text stream. The compiler proper must then tokenize this stream yet again for its own use. It is almost possible to do away with the double tokenization, but the strict definition of the C language does not really allow this. I tried it for the ANSI C compiler I wrote at work, and it is that area of the compiler that most needs redoing.

The sequence of tokens generated by the tokenizer must be parsed according to the grammar for the language. In most compilers, the parser also issues syntax-related error messages, attempts error recovery, and often does semantic (meaning) checking as well. These activities are all done by the Toy parsing routines. Parsing the input consists of identifying the grammar rules that fit the input, and interpreting that input according to those rules. For example, the only grammar rule that the sequence:

```
if a < b then write("Hello\n"); fi;
```

fits is that for an **if** statement. The condition **<expression>** is the subsequence **a < b**, which itself breaks down through the expression grammar into **<id> <comp-op> <id>**. There are no **<elif>**s, and the body **<statement-list>** has only one **<statement>**, which is a **<write-statement>**.

At this point the compiler writer has another option. Should an internal data structure be built that describes the parsed program? Such an internal form can make certain kinds of optimizations easier. For example, common sub-expression analysis, which attempts to avoid recalculating an expression whose value was recently computed, usually operates by scanning such an internal structure (usually in tree form) for similar sub-trees. Also, such a representation can be used to isolate the target CPU dependencies and the input language dependencies into separate sections of the compiler. Newer compiler systems are able to use this technique to provide a single 'back end' (optimization and code generation) to several different 'front ends' (tokenization and parsing), thus drastically reducing the work needed to create an entire family of compilers. Lattice C appears to use a technique like this – the 'quad' files are a quad (four-element) representation of the parsed program. This theoretically allows them to use the same 'front-end' for their Amiga compiler as, say, for the MS-DOS compiler. The Toy compiler, which is designed to be small and easy to understand, does not do this. Instead, the parsing code makes direct calls to code-generation routines, which generate the code for the constructs being parsed. This, unfortunately, prevents a clean separation between the parsing and the code generation.

As you would expect, the Toy compiler has little in the way of optimization. What it does have is there mainly for illustrative purposes and simply consists of some checks for special cases that can easily be done better.

The final stage of compilation is that of code generation. This stage typically does a lot of bit fiddling to construct the target machine instructions. This is where the compiler writer curses the CPU designers for not making the machine any where near as orthogonal as they claim to have. In a global sense, the MC68000 *is* reasonably orthogonal – the form and coding of addressing modes is consistent, and the positioning of most instruction fields is consistent. There are a number of nuisances, however. A minor, but easily spotted, example is the encoding of the operand length, which uses a two-bit field in some instructions and a three-bit field in others. The MOVE instruction uses a two-bit field, but with a different encoding than that used in other instructions. The EOR instruction appears to follow the pattern of ADD, SUB, AND, and similar instructions, but it is actually missing many of the modes – the encodings for *those* are used for other, totally different, instructions. One of the winners on the MC68000 has to be the shift/rotate instruction. There are two forms of it, one operating on registers, and one operating on memory operands. The memory operand form is distinguished by a size field of binary 11. The register form has a bit that specifies whether another field is an explicit shift count or is the number of a data register containing the shift count. Another field, in different places in the two forms, says what kind of shift is being done – logical, arithmetic, rotate, or rotate with extend.

The code-generation stage of a compiler thus has to contend with whatever idiosyncrasies the target CPU has. It also has to contend with the limitations of the CPU. These include such things as the number of registers available, the special-purpose use of some registers (the dreaded Intel 8086 family is notorious for this), and the size of various instruction fields (such as a 16-bit signed displacement on some modes, but an 8-bit signed displacement on others).

Again, there are several approaches to code generation. Many of the newer compilers are table-driven – they have a table of the instructions and forms that the target CPU has, along with some kind of encoded description of what they do. The compiler then attempts to match the requirements of the structure it is trying to generate code for against the table of instructions, to find one that matches. For any but the most simple structures, there will be no match, so the compiler will resort to heuristics like loading operands into registers, then trying again. For the simpler cases, there might be more than one match, so the compiler tries to decide which is 'cheaper'.

Once more, the Toy compiler doesn't do any of this – it has simple, special purpose code for each thing it tries do. The form of these routines will be discussed in full in the article on Toy's code generation.

The final example

As a final example of what the Toy compiler does, I will now show the stages of compilation for a small program. This will be referenced by the later articles in order to illustrate details of the compilation process.

Here is the initial program source:

```
int N, Sum;

proc main()void:

   Sum := 0;
   readln(N);
   while N ~= 0 do
      Sum := Sum + N;
      readln(N);
   od;
   write("The sum is: ", Sum, "\n");
corp;
```

The tokenizer reduces the program to this sequence of tokens:

'int' <id "N"> ',' <id "Sum"> ';' 'proc' <id "main"> '(' ')' 'void' ':' <id "Sum"> ':=' <number 0> ';' 'readln' '(' <id "N"> ')' ';' 'while' <id "N"> '~=' <number 0> 'do' <id "Sum"> ':=' <id "Sum"> '+' <id "N"> ';' 'readln' '(' <id "N"> ')' ';' 'od' ';' 'write' '(' <string "The sum is: "> ',' <id "Sum"> ',' <string "\n"> ')' ';' 'corp' ';'

Note that the tokenizer has totally discarded all the pretty spacing and indentation. There are a very large number of intermediate parsings that could be shown. One somewhere in the middle is:

```
<type> <id-list> ';'
'proc' <id> '(' <empty> ')' <result-type> ':'
   <empty>

   <assignment-statement> ';'
   <readln-statement> ';'
   'while' <exression> 'do'
      <assignment-statement> ';'
      <readln-statement> ';' <empty>
   'od' ';'
   <write-statement> ';' <empty>
'corp' ';'
```

Note that two **<empty>**'s are needed to allow a semicolon after the last statement in a **<statement-sequence>** (before the **od** and **corp**). The assembler code generated for this program is:

```
0000: link.w   fp,#0
0004: movem.l  <d1,d2,d3,d4,d5,d6,d7>,-(sp)
0008: clr.w    d7
000a: move.w   d7,Sum
0010: jsr      toy_readLine
0016: jsr      toy_readInt
001c: move.w   d0,d7
001e: move.w   d7,N
   LOOP:
0024: move.w   N,d7
002a: clr.w    d6
002c: cmp.w    d6,d7
002e: sne      d7
0030: andi.w   #1,d7
0034: tst.w    d7
0036: beq      LOOPEXIT
003a: move.w   Sum,d7
0040: move.w   N,d6
0046: add.w    d6,d7
0048: move.w   d7,Sum
004e: jsr      toy_readLine
0054: jsr      toy_readInt
005a: move.w   d0,d7
005c: move.w   d7,N
0062: bra      LOOP
   LOOPEXIT:
0064: jsr      16(pc)
0068: "The sum is: "
0076: jsr      toy_printString
007c: move.w   Sum,d7
0082: move.w   d7,-(sp)
0084: jsr      toy_printInt
008a: jsr      4(pc)
008e: "\n"
0090: jsr      toy_printString
0096: movem.l  (sp)+,<d7,d6,d5,d4,d3,d2,d1>
009a: unlk     fp
009c: rts
```

Finally, the AmigaDOS object module hunks produced are:

```
hunk_code:
0000: 4e56 0000 48e7 7f00 4247 33c7 0000 0116 *NV..H...BG3.....*
0010: 4eb9 0000 0000 4eb9 0000 0000 3e00 33c7 *N.....N.....>.3.*
0020: 0000 0114 3e39 0000 0114 4246 be46 56c7 *....>9....BF.FV.*
0030: 0247 0001 4a47 6700 002c 3e39 0000 0116 *.G..JGg..,>9....*
0040: 3c39 0000 0114 de46 33c7 0000 0116 4eb9 *<9.....F3.....N.*
0050: 0000 0000 4eb9 0000 0000 3e00 33c7 0000 *....N.....>.3...*
0060: 0114 60c0 4eba 0010 5468 6520 7375 6d20 *..`.N...The sum *
0070: 6973 3a20 0000 4eb9 0000 0000 3e39 0000 *is: ..N.....>9..*
0080: 0116 3f07 4eb9 0000 0000 4eba 0004 0a00 *..?.N.....N.....*
0090: 4eb9 0000 0000 4cdf 00fe 4e5e 4e75 0000 *N.....L...N^Nu..*
hunk_reloc32:
   hunk 3: 134
   hunk 2: 146 120
   hunk 5: 86 24
   hunk 4: 80 18
   hunk 1: 126 94 74 66 60 38 32 12
hunk_end
```

The actual executable object file created contains the Toy start-up and run-time library before these hunks, but they are fixed for all Toy programs, so they are not shown here. About the only part of the original program that is still recognizable to a human reader is the string "The sum is: ".

Next month

The next article in this series will examine the data structures used by the Toy compiler and will present the tokenizer and the symbol table handler in detail.

Listing 1: hanoi.t

```
/*
 * hanoi.t - standard "Towers of Hanoi" in toy.
 */

proc printPeg(int peg)void:

    if peg = 1 then
        write("left");
    elif peg = 2 then
        write("center");
    else
        write("right");
    fi;
corp;

proc hanoi(int n, fromPeg, usingPeg, toPeg)void:

    if n ~= 0 then
        hanoi(n - 1, fromPeg, toPeg, usingPeg);
        write("Move disk from ");
        printPeg(fromPeg);
        write(" peg to ");
        printPeg(toPeg);
        write(" peg.\n");
        hanoi(n - 1, usingPeg, fromPeg, toPeg);
    fi;
corp;

proc main()void:
    int n;

    write("How many pegs? ");
    readln(n);
    hanoi(n, 1, 2, 3);
corp;
```

Listing 2: shoot.t

```
/*
 * shoot.t: a simple, character graphic, target shoot game in Toy.
 */

int
    LINE_COUNT,         /* number of text lines on screen */
    STATUS_LINE,        /* line we can put status info on */
    MESSAGE_COLUMN,     /* the column for messages */
    PROMPT_COLUMN,      /* column for input prompt */
    MAXIMUM_COLUMN,     /* maximum column for display */
    SCALE,              /* scale factor for fixed-point */
    HORIZONTAL_SPEED;   /* speed of shot in horizontal direction */

int
    Seed,               /* seed for random number generator */
    Score,              /* the current number of shots taken */
    GunColumn,          /* column the gun is in */
    TargetColumn;       /* column the target is in */

bool
    MessageShown;       /* true => a message is currently displayed */

/*
 * random - return a pseudo-random number in the given range.
 */

proc random(int rang)int:

    Seed := Seed * 2137 + 173;
    if Seed < 0 then
        Seed := -Seed;
    fi;
    Seed - Seed / rang * rang
corp;

/*
 * clearScreen - clear the display screen.
 */

proc clearScreen()void:

    write("\(12)");
corp;

/*
 * disableCursor - turn the cursor off - makes output faster.
 */

proc disableCursor()void:

    write("\(155)0 p");
corp;

/*
 * enableCursor - turn the cursor on.
 *      NOTE: there seems to be a minor bug in the Amiga's CON: handler here.
 *      It isn't supposed to matter whether an output control sequence uses
 *      the CSI (decimal 155) or ESC-[ to start a control sequence. However,
 *      using the CSI mode here works if the program is compiled with toy,
 *      but not if it's compiled with Draco - probably relating to the fact
 *      that the Draco run-time system buffers the output.
 */

proc enableCursor()void:

    write("\(27)[ p");
corp;

/*
 * moveTo - move the cursor to the specified row and column.
 */

proc moveTo(int row, col)void:

    write("\(155)", row, ";", col, "H");
corp;

/*
 * setSpecial2 - set the character display to boldface, colour 2.
 */

proc setSpecial2()void:

    write("\(155)1;32m");
corp;

/*
 * setSpecial3 - set the character display to boldface, colour 3.
 */

proc setSpecial3()void:

    write("\(155)1;33m");
corp;

/*
 * setNormal - reset character display back to normal.
 */

proc setNormal()void:

    write("\(155)0m");
corp;

/*
 * writeScore - write the current score to the status line.
 */

proc writeScore()void:

    moveTo(STATUS_LINE, 8);
    write(Score);
corp;
```

```
/*
 * drawGround - draw the ground as normal colour '_'s.
 */

proc drawGround()void:
    int i;

    moveTo(STATUS_LINE - 1, 1);
    i := 1;
    while i < MAXIMUM_COLUMN do
        i := i + 1;
        write("-");
    od;
corp;

/*
 * drawGun - draw the gun at the proper position.
 */

proc drawGun()void:

    moveTo(STATUS_LINE - 4, GunColumn - 2);
    write("+-I-+");
    moveTo(STATUS_LINE - 3, GunColumn - 2);
    write("|   |");
    moveTo(STATUS_LINE - 2, GunColumn - 2);
    write("|   |");
    moveTo(STATUS_LINE - 1, GunColumn - 2);
    write("$$$$$");
corp;

/*
 * clearMessage - clear any message now shown.
 */

proc clearMessage()void:

    if MessageShown then
        MessageShown := false;
        moveTo(STATUS_LINE, MESSAGE_COLUMN);
        write("                    ");
    fi;
corp;

/*
 * drawTarget - draw the target at the proper position.
 */

proc drawTarget()void:

    moveTo(STATUS_LINE - 4, TargetColumn);
    write("^");
    moveTo(STATUS_LINE - 3, TargetColumn - 1);
    write("< >");
    moveTo(STATUS_LINE - 2, TargetColumn);
    write("V");
corp;

/*
 * missed - issue "Missed!" as a message.
 */

proc missed()void:

    moveTo(STATUS_LINE, MESSAGE_COLUMN);
    setSpecial3();
    write("Missed!");
    setNormal();
    MessageShown := true;
corp;

/*
 * hitTarget - fire a shell and see if it hits. Return true if it does.
 */

proc hitTarget(int speed)bool:
    int row, col, oldRow, oldCol;
    bool notDone, hit;

    setSpecial2();
    row := (STATUS_LINE - 5) * SCALE;
    col := GunColumn * SCALE;
    hit := false;
    notDone := true;
    while notDone do
        if row / SCALE > 0 then
            moveTo(row / SCALE, col / SCALE);
            write("*");
        fi;
        oldRow := row;
        oldCol := col;
        col := col + HORIZONTAL_SPEED;
        row := row - speed;
        speed := speed - 1;
        if oldRow / SCALE > 0 then
            moveTo(oldRow / SCALE, oldCol / SCALE);
        else
            moveTo(0, 0);
        fi;
        write(" ");
        if row / SCALE > 0 then
            if row / SCALE < STATUS_LINE - 1 then
                moveTo(row / SCALE, col / SCALE);
                write("*");
            else
                moveTo(0, 0);
                write(" ");
            fi;
        fi;
        if row / SCALE >= STATUS_LINE - 1 then
            /* shell has hit the ground */
            moveTo(STATUS_LINE - 2, col / SCALE - 2);
            write("BOOOM");
            missed();
            notDone := false;
        elif col / SCALE >= MAXIMUM_COLUMN - 1 then
            /* shell has gone off the right of the screen */
            moveTo(row / SCALE, col / SCALE);
            write(" ");
            missed();
            notDone := false;
        elif col / SCALE = TargetColumn then
            if row / SCALE = (STATUS_LINE - 3) then
                /* hit the target! */
                hit := true;
                drawTarget();
                moveTo(STATUS_LINE, MESSAGE_COLUMN);
                setSpecial3();
                write("Hit!");
                setNormal();
                MessageShown := true;
                notDone := false;
            fi;
        fi;
    od;
    hit
corp;

/*
 * getSpeed - prompt for and read the next speed value.
 */

proc getSpeed()int:
    int speed;

    moveTo(STATUS_LINE, PROMPT_COLUMN);
    write("speed (1-99): ");
    enableCursor();
```

```
    readln(speed);
    disableCursor();
    moveTo(STATUS_LINE, PROMPT_COLUMN);
    write("                    ");
    clearMessage();
    speed
corp;

proc main()void:
    int speed;
    bool notDone;

    MESSAGE_COLUMN := 15;
    PROMPT_COLUMN := 58;
    MAXIMUM_COLUMN := 76;
    SCALE := 100;
    HORIZONTAL_SPEED := 40;
    MessageShown := false;
    while
        write("How many text lines on your screen? ");
        readln(LINE_COUNT);
        notDone := LINE_COUNT < 10;
        if LINE_COUNT > 50 then
            notDone := true;
        fi;
        notDone
    do
        write("Value out of range (10 - 50), try again.\n");
    od;
    STATUS_LINE := LINE_COUNT - 1;
    write("Enter random number seed: ");
    readln(Seed);
    disableCursor();
    clearScreen();
    GunColumn := 4 + random(10);
    TargetColumn := 30 + random(40);
    setSpecial3();
    drawGround();
    drawGun();
    drawTarget();
    setNormal();
    Score := 0;
    moveTo(STATUS_LINE, 0);
    write("Score: ");
    writeScore();
    notDone := true;
    while notDone do
        speed := getSpeed();
        if speed > 0 then
            if speed >= 100 then
                moveTo(STATUS_LINE, MESSAGE_COLUMN);
                setSpecial3();
                write("Too large.");
                setNormal();
                MessageShown := true;
            else
                if hitTarget(speed) then
                    notDone := false;
                else
                    setSpecial3();
                    drawTarget();
                    setNormal();
                fi;
                Score := Score + 1;
                writeScore();
            fi;
        else
            notDone := false;
        fi;
    od;
    enableCursor();
    moveTo(STATUS_LINE + 1, 1);
corp;
```

. . . if you have at least 3 years of assembly language programming experience, preferably one year of 68000 or have an NTSC/PAL Video Hardware background. Join the team that makes no compromises and help us change the world. Key positions available at the fastest growing Amiga company. Hackers call Stephanie at 913-354-1146 or write to Stephanie at 115 West Crane Street, Topeka KS 66603.

Schedule Assistant

Voice and Sound Reminders -
Create your own or use NAG PLUS library.

Perpetual Calendar *-Enter 198 events per day.*

Auto Dialer *-Connects YOU, not your modem.*

AREXX Port *-Commands any timed event or action.*

Notepad *-Click on any word to open text editor.*

Print • Search • And Much More.

Suggested Retail $79.95. Ask your dealer or contact:

Gramma Software
17730 15th Avenue N.E.
Suite 223
Seattle, Washington 98155
Phone #: (206) 363-6417
Fax #: (206) 361-0429

Hard Copy

Turbo Silver tarnished by meagre manual

by Tim Grantham

Turbo Silver is a 3D modelling and animation program from Impulse Inc. that has a lot going for it. It features fast, multiple-mode renderings; a rich selection of object attributes (colour, reflectivity, index of refraction, and so on); hierarchically-structured objects; a flexible animation editor with localized looping and stereo image support; 68020/68881 support; and many other capabilities in a package costing less than $200 US. It is relatively bug-free, is constantly being improved and is blessed with a publisher that provides solid customer support in the form of a toll-free phone number (in the USA only) and a section in the AMIGAVENDOR forum on Compuserve.

Unfortunately, the program has a millstone round its neck: a manual that has violated the first law of software documentation: "What does the user need to know?"

Who *are* the users of Turbo Silver? According to Stan Kalisher, Product Director at Impulse, they range from novices to professionals, teenagers to senior citizens: in other words, the target market is anyone with an Amiga and an interest in 3D computer graphics.

This is, admittedly, a real challenge for the manual writer. The author must not only provide a reference for the features of the program; he or she must also provide instruction on the subject of 3D modelling and animation. Without the former, the advanced user will be seriously hampered in his or her quest to make efficient use of the program; without the latter, the novice will not even know where to start.

Creating 3D animations

The creation of 3D animated graphics on a computer basically breaks down into three related tasks: modelling (constructing) objects; rendering (drawing) them; and animating (moving) them.

Programs like Turbo Silver and Sculpt-Animate 4D integrate these tasks into one program; alternatively, you can create and render objects in a program like Aegis' Modeller 3D and animate them in the same company's animation program VideoScape 3D.

Objects typically consist of vertices, edges and faces. (See previous articles by Eric and Cathryn Graham for more information.) Object-creation programs provide tools to construct objects, edit them, clone them and so on. Objects can be clustered with other objects to form a group that can be manipulated as a single, more complex object.

Sophisticated modellers like Turbo Silver and Sculpt-Animate 3D also provide another key feature: object hierarchies. With this capability, an object can be defined to be a parent object. Any object that is then defined as a child of that object will be affected by a change made to the parent object. This is essential in the modelling of articulated objects, such as a robot arm. If you wanted to move the elbow of the robot arm, you would want the attached child objects – the wrist, palm, fingers and so on – to follow, without having to laboriously specify the motion of each individual child object. Contrariwise, if you wanted to flick the finger of the robot hand, you would not necessarily want that movement to affect the elbow.

While object hierarchies add more power to the modelling program, they also make the program more complex. Simple selection of an object becomes much more complicated: does clicking on a vertex of an object select the vertex, the associated edges, the associated faces, the entire object, the attached objects in the group, the child objects (or child grouped objects) or the parent (or grandparent) object? (The discussion provided in the Turbo Silver manual on this subject is so terse as to be incomprehensible.)

To further complicate matters, each object (even each face in an object) can have many attributes, or characteristics: colour, reflectivity, roughness, specularity, index of refraction and a host of others. These characteristics determine the appearance of an object when it is rendered, as does the means of rendering. Turbo Silver provides three rendering methods: wireframe, surface modelling and ray tracing.

If objects are to be animated, another dimension enters the picture: time. Time can be modelled in the form of motion paths. While these do not usually have such attributes as colour or reflectivity, they can be hierarchically connected and manipulated much as other objects are.

By the time you begin to use such effects as *tweening* (where the computer generates an in-between state of an object in motion, given two boundary states) and nested looping of frames, you have begun to enter the realm of the professional 3D graphics artists. Any manual that purports to provide a guide to such a complex world must be more than a mere map of the territory: it should be a substantial volume written by someone who's been there, who knows the perils and the pitfalls as well as the quickest route to where you want to go.

How *not* to write a manual

I am emphatically not a computer graphics professional. I have read a couple of introductory books on the principles of computer graphics; I know what a B-spline is, I know something about Gourard and Phong shading. But I have no practical experience with the methods, nor how such methods are usually incorporated into such a feature-rich program as Turbo Silver. When I was first confronted with the Turbo Silver screen, I really didn't know where to begin. All I could do was count on my own intelligence and hope that the manual had a thorough tutorial.

The Turbo Silver manual has no tutorial at all. It does have what it calls a 'Quick Start' section intended to introduce the reader to the program. This section fails because it is simply a recipe... 'Do this; then, do that...' that does not provide any explanation of the principles of 3D modelling and animation. It simply reassures the user that the program actually does something; the novice user probably has no idea what that is.

Other examples of this recipe approach to instruction are scattered throughout the manual to illustrate various features. Each suffers to a greater or lesser degree from the same affliction: a lack of context. The text is accurate enough and, *once you have a thorough understanding of the program*, it is readable. But the very point of a manual is to provide a foundation for understanding, to *reduce* the need for trial and error experimentation.

As a relative tyro, it took me many hours of experimentation with the program (during which I repeatedly lost much of what I was working on) and multiple read-throughs of the manual before I felt I had a reasonable grasp of the principles of operation of Turbo Silver.

Nor can I now, as at least an intermediate user, easily use the manual as a reference. While there is a comprehensive table of contents, there is no index. Nor is there any hierarchy of sections: that is, no numbering scheme or variation in the size of subheads to indicate that a description of a sub-menu item is a smaller part of a section on the main menu.

The problem is compounded by the placing of examples of animation functions (the animation by 'Story' function, for example) within the section on object creation functions, without any page numbers at least to cross-reference the example to the appropriate part of the animation section.

When I spoke with Kalisher, he said, "We develop these products primarily for our own use. Then if we think other people are interested in them, we bring them to market."

Therein, methinks, lies the root cause of the poor manual: it *reads* like an in-house reference written for skilled operators, as opposed to a guide for what is essentially a mass-market product to be used mostly by people unfamiliar with what such a program does.

As it is now, it's simply a design specification for the program – it's actually written as a listing of each of the menu selections, with an explanation of what each one does. If Impulse wasn't so good about its customer support, I'd say that the manual betrays an almost arrogant disregard for the needs of the average purchaser of Turbo Silver.

How to write a better manual

How can it be improved? With only a modest amount of work, including the addition of a complete, cross-referenced index, and a more complete section on the program's 'virtual file storage' (see below), the manual for Turbo Silver could become at least an acceptable reference. However, to be useful to those of us who know nothing about 3D modeling and animation, the manual should also provide separate chapters and tutorials on:

- a brief history of 3D modelling and animation on computers
- object creation and editing (including a tutorial on hierarchically-related objects)
- object attributes and their effects on the final image
- the use of paths in both modelling and animation
- the assembly of complete animations from multiple scenes

It would also be nice to have chapters devoted to the use of the powerful 'Story' functions, the rôle of the key cell, and a collection of 'hints'.

The manual also needs a chapter on the program's 'virtual file storage'. This is a data storage system that automatically creates cell, rendering and animation directories and files, and automatically updates them as you make changes.

The problems with this system develop when you *don't* want files updated (perhaps you made a mistake in the creation of an object). Unless you remember to specifically invoke the Undo and Escape functions before exiting the object editor, Turbo Silver will overwrite the original data.

Yet neither function is mentioned in 'Quick Start'. The virtual file storage capability, while definitely useful to the experienced user, bit this novice several times during the creation of his first substantial piece of animation.

The Silver lining

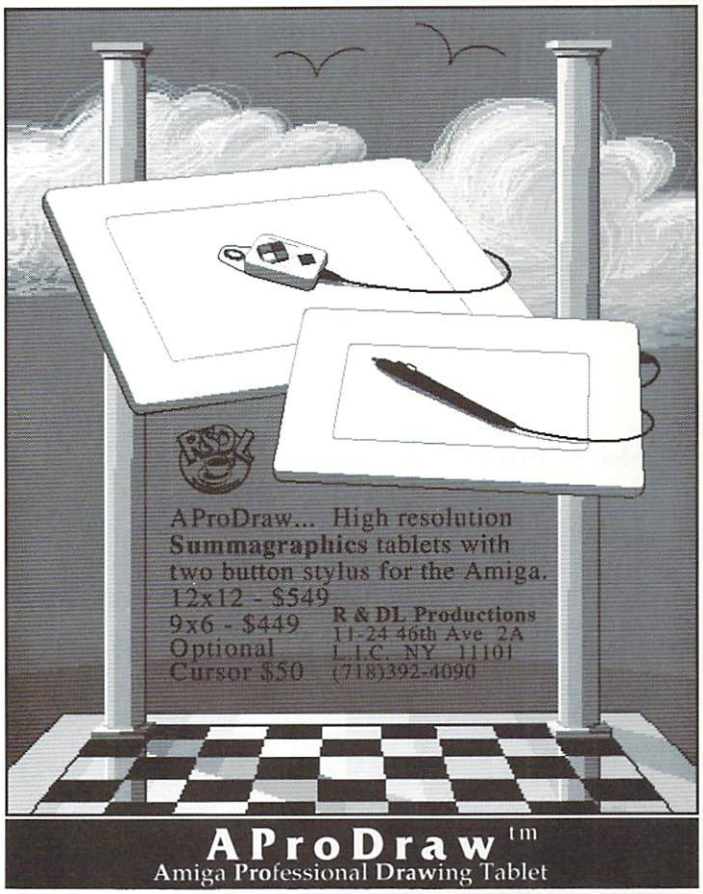
The manual does have some good points. It has been nicely typeset and laser-printed, and comes in a three-ring binder. The binder will make it easy to add updates (Kalisher assured me that updates *would* be forthcoming) and lets the manual lie flat for easy computer-side reference. The manual's spelling, grammar and syntax are fine, as is the aforementioned table of contents.

Impulse has also made an hour-long instructional videotape available – a laudable action. The tape was actually put together by one of the company's users of Turbo Silver, so it will likely provide the kind of user perspective the manual so desperately needs. The tape costs $24.95 US and is available from Impulse Inc, 6870 Shingle Creek Pkwy., Unit #12, Minneapolis, MN 55430, USA. Telephone (800) 328-0184. In Minnesota and outside the USA, call (612) 566-0221.

Kalisher told me that, ideally, Impulse would rather simply provide the videotape to buyers and dispense with the manual altogether. I hope Impulse doesn't do this. Sure, the videotape is a great idea but, firstly, it presupposes that the customer has a VCR; and secondly, it's not a random access device. You could spend hours with your finger on fast forward trying to find more info on the one function you need to complete your epic animation.

If I could be convinced that the choice was Turbo Silver with a poor manual or no Turbo Silver at all, I would probably take the former. There is no question that producing the kind of manual that Turbo Silver deserves would be a fairly expensive proposition. That doesn't make it less necessary. But Impulse obviously believes the old saw, "It's not documentation that sells a program; it's *features*."

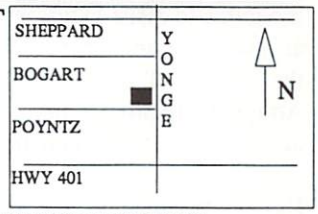
To which I can only reply, "What good are the features if you can't find out how to use them?"



CANADA'S BEST SELECTION OF
AMIGA - C64 - C128 SOFTWARE
COMSPEC HARD DRIVES AND BERNOULLI DRIVES
AVAILABLE FOR
THE AMIGA 500 AND 2000

Electronics 2001
COMPUTER PRODUCT CENTRE

4736 YONGE STREET
WILLOWDALE ,ONT
M2N 5M6
TEL: (416) 229-2700



AUTHORIZED COMMODORE DEALER SINCE 1979

New Store!

Fisher's
Computers & Software
Authorized Amiga Dealer

8005 Archibald
Rancho Cucamonga, CA 91730
(714) 987-1662

Serving The Inland Impire

Let There Be LightBox!

A tool for traditional animators

Review by Derek Grime

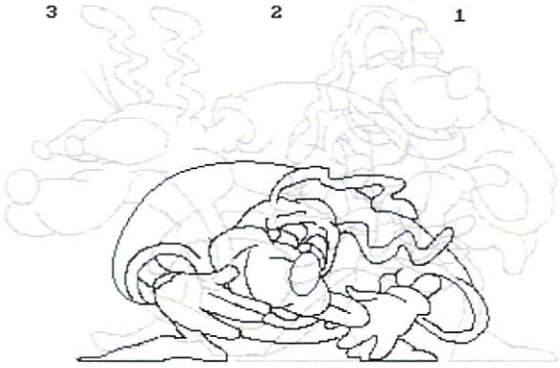
LightBox: animation tool

From R & DL Productions, 11-24 46th Avenue #2A, Long Island City, NY 11101, Phone (718) 392-4090

Suggested retail: $189.95 US

Classical animation is back. *Who Framed Roger Rabbit?* and the re-release of *Bambi* both made a bundle at the box office last year. This type of full character animation remains the most entertaining to watch, but is by far the most difficult to create.

Animators who use the Amiga have a range of programs available that will display their drawings in real time. Page Flippers have become best sellers, yet until now there haven't been any programs to assist in the actual process of animation. With the release of *LightBox*, the Amiga is finally coming of age for the classical animator.



Here's the impasse animators face when using the Amiga. Keyframe character animation means that the artist first draws the extremes of a movement. The move is then fleshed out with a series of in-between drawings. Traditionally, the extremes are placed on a light table and an in-between is traced between the two. Then the in-between is in-betweened. This continues (and continues...) until enough drawings are added to smooth out the motion.

Amiga animators I know use two methods for in-betweening. Some draw the images on paper and then scan them in frame by frame. There is a lot of clean-up work involved. Others use a drawing tablet to trace the pictures into a paint program directly. No method has existed that used the Amiga to assist in the in-betweening process.

That is, until LightBox came along. After drawing in the keyframes, LightBox shows you the two poses on the same screen in light colors. You in-between in black on top of the extremes. Three frames are on the screen at once, yet each drawing exists as a separate entity. At any time, new drawings can be added between any two other frames. It's an elegant solution to a difficult problem.

LightBox is keypad driven. The '4' key moves you back one drawing in the sequence, the '6' moves one forward and the '5' flips through the action in real time. All functions have keyboard equivalents.

A few tools are built into LightBox. The drawing functions are rather limited, giving you freehand, dotted or custom brushes. There are no tools for arcs, circles or lines.

Resolution is limited to lo-res with one color. There is no overscan mode.

LightBox also has trouble taking input from the Easyl drawing tablet. As I've never encountered difficulties with the tablet in the past three years, I'll assume that the software is playing tricks with the mouse events.

I talked with the nice people at R & DL productions and they are in the process of addressing all these issues. An upgrade with more features is planned for the near future, and an upgrade path will be announced for registered users.

LightBox is a godsend to the classical animator. It does not in-between for you, but makes the process as painless as possible. The lack of tools is a minor issue, as it's really meant to be used in conjunction with your favourite paint program. Flying 3D logos is fun, but if you want to some day be a *real* animator in the Tex Avery or Bob Clampett style, give LightBox a try ... and just think of all the money you'll save on paper!

FREE: Amiga Demos

Have you ever wished you could view a program before you purchased it? Do artist conceptions and written descriptions really give you a good idea about the program you are buying? No!

We will send you actual manufacturers demonstration programs for you to view, on your own computer, in your own time. The gimmick is: if you like the program, buy it from us! We're even competitively priced!

Send us your name and address, we'll send you an ***Amiga Demo FREE!***

We ship anywhere in the World!

Forethought
Innovatiove Software Distribution
P.O. Box 1221
Lewiston, NY 14092

68000 DISASSEMBLY

OTG Software

ANNOUNCING . . .
DSM VERSION 1.0d

DSM is a full-featured disassembler for the Amiga. Check out these features and you'll see why programmers agree, "DSM is the best disassembler currently available for the Amiga, bar none."

- DSM disassembles any Amiga program that does not use overlays.
- DSM produces assembler-ready output that is 100% compatible with the Amiga assembler, assem. Excellent results can also be obtained using the C.A.P.E. assembler from Inovatronics.
- DSM features true text detection.
- Improved disassembly is supported using the built-in expert system mode.

If your local Amiga dealer doesn't carry DSM, send check or money order to:

OTG Software
200 West 7th Street Suite 618
Fort Worth, TX 76102

TX residents add 7.25% sales tax.

Price
$67.50 US

COLOR RIBBONS & PAPER

RIBBONS: Red, Blue, Grn., Brwn., Purple, Yel.

Ribbons Price Each	Black	Color	Heat Transfer
Brother M1109	4.95	5.95	7.00
C. Itoh Prowriter Jr.	7.00	9.00	–
Citizen 120D/180D	5.00	6.00	7.95
Commodore MPS 801	4.50	5.25	5.75
- MPS 802/1526	6.25	7.25	–
- MPS 803	4.95	5.95	7.00
- MPS 1000	3.95	4.95	6.75
- MPS 1200/1250	5.00	6.00	7.95
- 1525	6.50	8.00	–
Epson MX80/LX800	3.75	4.25	6.75
Okidata 82/92	1.75	2.25	4.50
Okidata 182/192	6.50	7.50	–
Panasonic K-XP 1080	6.75	7.75	–
Seikosha SP 800/1000	5.25	6.50	7.95
Star SG10	1.75	2.25	4.50
Star NX10/NL10	5.00	6.00	7.95
Star NX1000	4.50	5.50	8.00
Star NX1000C – 4C	–	10.75	–

COLOR PAPER

BRIGHT PACK – 200 Sheets/50 each color: Red, Blue, Green, Yellow. 9 1/2 x 11 – $11.90/pk.

PASTEL PACK – 200 Sheets/50 each color: Pink, Yellow, Blue, Ivory. 9 1/2 x 11 – $11.90/pk.

T-SHIRT RIBBONS (Heat Transfer) – Red, Blue, Grn., Brwn., Purple, Yel., Blk. Call For Price & Availability.

COLOR BANNER BAND PAPER – 45 ft./roll – $9.95/ea.

For ribbons & paper not listed above, call for price & avail. Price & spec. subject to change w/o notice. Min. order $25.00. Min. S&H $3.50 min. Visa, MC, COD.

RAMCO COMPUTER SUPPLIES
P.O. Box 475, Manteno, IL 60950 U.S.A.
1-800-522-6922 • 815-468-8081

A CSA TURBO 68020 FOR ONLY $399?

Hard to believe isn't it? CSA broke the price barrier in 32 bit technology. Now you don't have to settle for a far less capable 68000 accelerator, you *can* have affordability, capability, and speed in one easily installed package.

"1000% faster than my stock Amiga™?"

CSA's new 68020 *Midger Racer*™ flies through MaxiPlan™ spreadsheets over 60% faster than the standard Amiga™, and over 50% faster than 68000 based accelerators. With the addition of a 12 MHz 68881 co-processor, the *Midget Racer*™ can crunch 2 Million FLOPS over 1000% faster than a stock Amiga™.

"600% faster than a 68000 accelerator?"

In fact, programs like Sculpt & Animate 3D or 4D™, Draw 2000™, and X-CAD™ have been written to directly access the 68020 & 68881, and may not even *run* with a 68000 based accelerator. And with a 12 MHz 68881, CSA's *Midget Racer*™ is up to 600% faster than a 68000/68881!

"All that for only $399?"

No, there's more! It's also easily installed into ***any*** Amiga, supports an '881/2 up to 33 MHz, and it's available now. Don't settle for second best, call your Amiga dealer or CSA today!

CSA. We set the standards in performance.

CSA

Computer System Associates, Inc.
7564 Trade Street
San Diego, CA 92121
619-566-3911 FAX: 619-566-0581

Trademarks: Amiga: Commodore-Amiga, Inc.; MaxiPlan: Oxxi, Inc.; 68000/020/881/882: Motorola, Inc.; Sculpt & Animate 3D/4D: Byte by Byte, Inc.; Draw 2000: Aegis, Inc.; X-CAD: Taurus, Inc.; Midget Racer: Computer System Associates, Inc.

GFA BASIC 3.0

Review by Chris Zamara

GFA BASIC 3.0: BASIC language editor/interpreter

Published in the U.S.A. by Antic Software, 544 Second Street, San Francisco, CA 94107. Phone (415) 957-0886

Suggested retail: $139.95 U.S.

GFA BASIC is an integrated editor/interpreter system for BASIC program development. A glance through the thick manual will tell you right away that the language is very rich, with hundreds of commands and functions ranging from general programming to graphics, sound and animation routines. The editor is language-sensitive and quite fast. The interpreter itself is also very fast. And with all this good news comes some bad news as well.

It is unfortunate that you will encounter what is perhaps the system's weakest point immediately - the manual. It is almost devoid of any tutorial or useful examples, and launches into the descriptions of the commands almost immediately. The command descriptions appear to be taken from notes scribbled down by the programmer, then poorly translated to English from the original German. Some of this text can be quite humorous (the syntax for the OCT$ function is shown as OCTOBER$(x[,y]), for example), and while an expert Amiga programmer may be able to figure out what they mean, a beginner just learning the system will be in an unfortunate position. At best, the descriptions are difficult to understand, but at worst they are just plain incorrect. How about the first paragraph in the description of the INLINE command:

> INLINE adr,num
>
> Remarks are not allowed directly behind this command because the area normally used for a remark is the exact location that the desired memory location is reserved at. Initially the memory area is erased (filled with zeros). This memory area always begins at an even address. When INLINE is executed the address of this integer variable is written to adr. When the program is saved or loaded this memory area is saved or loaded with the program.

Huh? As it turns out, INLINE is a useful device that lets you incorporate binary data into the BASIC program itself and access that memory with various commands. You'd never know it from the description, however.

Terms like ViewPort, DMA, clipping, and IDCMP are thrown around with abandon, which can be disconcerting to those who turned to BASIC in order to avoid learning the Amiga at that level in the first place. The sheer number of commands can be daunting in itself, but the explanations make learning the language even harder.

It would be unfair to dwell further on the manual, so let's assume that you're familiar enough with the Amiga and other BASICs to understand it. Perhaps the best thing about GFA is the editor and overall programming environment, the very thing that makes the standard AmigaBasic almost intolerable. The editor is reasonably easy to use once you learn a few of the important keystrokes, and it scrolls, cuts and pastes quite quickly. Statements automatically indent to the appropriate nesting levels based on the current state of IF, WHILE, FOR and other control statements; this makes it easy to spot mismatched END statements immediately. A degree of 'outline processing' lets you shrink an entire procedure to a single line instantly, and expand it at will. This lets you concentrate on the code you're currently writing, and consider the already-written procedures as 'black box' modules. Program lines are checked for syntax as they are entered, and keywords are converted to uppercase automatically. Some syntax is also corrected to GFA's preferred form: for example, 'endproc' will change into 'RETURN' after you enter the line.

Common editing and control operations (run, trace, etc.) are available from function keys or via gadgets at the top of the screen. A 'direct mode' is available, which opens a small window for direct examination of variables and other single-line commands. The smoothness of the editor and environment makes it easy to develop programs, provided you are comfortable with the language.

The language itself is huge. It seems to have every feature you'd ever need, and many that you probably don't. It was obviously developed by hard-core assembly language programmers with an urge to get right down to the machine level. Variables can be declared as type Boolean, Byte, Word, Integer, Float or String. There are assembler-like commands to manipulate memory bits, bytes and words directly, commands for allocating and freeing memory, methods for calling C and assembler routines directly, and even a way to directly read from and write to CPU registers! On the other end of the spectrum, there are some convenient high-level calls not found in other BASICs, like a file requester, sort routines, and a screen dump to printer.

(concluded on page 80)

News LINK

Three new Amigas: Commodore International has announced the availability of two new models in the Amiga 2000 product line, and introduced an Amiga that runs under the increasingly popular UNIX operating system.

The *Amiga 2000HD* comes standard with a 40-Mb, 3.5-in., 28 millisecond autoboot hard disk drive configured with Workbench 1.3, 1 Mb of RAM and 1 internal 3.5-in floppy disk drive. The *Amiga 2500* provides the same drive configuration but also features the *A2620* card installed in the machine's CPU slot. This card has a 68020 processor running at 14.3 MHz, a 68881 math coprocessor, a 68851 memory management unit, 2 Mb of 32-bit RAM (upgradeable to 4 Mb of 32-bit RAM), UNIX boot ROMs and 1 Mb of 16-bit 'chip' RAM.

The *A2500UX* uses the UNIX ROMs on the A2620 card to boot UNIX System V, version 3.1 from an integral 80-Mb, 19 millisecond hard drive. (An internal 150-Mb streaming tape drive is included for archiving and routine backup procedures.) The system comes with 4 Mb of 32-bit RAM and 1 Mb of 16-bit 'chip' RAM.

The UNIX operating system, developed by AT&T Laboratories, is now enjoying broad acceptance. UNIX is intended to achieve portability of software across a variety of hardware platforms. Implementations of UNIX run on everything from Cray supercomputers to IBM PC ATs. Amiga UNIX (AMIX) represents the latest version of UNIX, System V, Release 3.1, which includes text processing and software development tools that include the Documentor's Workbench, device-independent *troff*, *eqn*, *tbl* and a C compiler developed by Motorola specifically for the 68020/68881.

A proprietary windowing system is incorporated within the AMIX environment; as with AmigaDOS, it uses the custom graphics hardware to manage the display, freeing the CPU for other tasks. Separate 'screens' are supported, as in AmigaDOS, permitting multiple windowing systems. While the A2500UX does not run AmigaDOS and UNIX simultaneously, Workbench 1.3 is installed on the hard disk and the user can select which operating system to use at boot time.

It is expected that Commodore, upon release of the system, will provide support for X-Windows and BSD (Berkeley Software Development) enhancements for networking, and will provide an upgrade path to a 68030-based system running at 25 MHz. Beta versions of the AMIX system are currently being made available to developers; it should be available to the public later this year. For more information, contact Commodore International, 1200 Wilson Drive, West Chester, PA 19380, USA. (215) 431-9100. In Canada, contact Commodore Business Machines, Ltd., 3470 Pharmacy Ave., Agincourt, ON M1W 3G3. (416) 499-4292.

Amiga X11 released: GfxBase's *X11* implementation for the Amiga is now officially available. X11 on the Amiga takes advantage of the custom blitter chip to speed operations, providing performance said to match or exceed X11 machines costing much more.

X11 on the Amiga uses TCP/IP over Ethernet for connection to other machines, and a shared memory message passing scheme for local connections. It includes several clients that run under AmigaDOS, such as *uwm*, *xcalc* and *bitmap*. GfxBase says that the use of local clients reduces network bandwidth requirements compared to simple X terminals. (*Xlib* and *Xt* libraries are planned for introduction later this year.) The GfxBase version of *xpr* supports up to fifty different printers, enabling the Amiga to act as a general purpose printer server. Supported printers include the Howtek PixelMaster, the HP PaintJet and the Calcomp Colormaster. An optional IEEE-488 interface is available.

Amiga X11 has been successfully connected to computers from DEC, Sun Microsystems, Apollo Computers, Hewlett-Packard and others at the X Connectathon in February, 1989. Cost of X11 is $395 US; adding a three-button optical mouse brings the cost to $475 US; and a package that includes an Ethernet board for the A2000 is priced at $1339 US. For more information, contact GfxBase, 1881 Ellwell Dr., Milpitas, CA 95035, USA. (408) 262-1469

Yet another Modula-2 compiler: M2S, Inc. has introduced *M2Sprint*, a Modula-2 development system. The system includes: a single-pass Modula-2 compiler, a text editor, a single-pass linker, a symbolic debugger, program profiler, full Amiga interface library, Modula-2 library, C-style I/O library, IFF library, ARP interface library, ARexx interface library, full source to library modules and demo programs, and ring-bound programmer's guide and library reference manual. The editor can load and keep resident the compiler and linker.

The M2Sprint compiler can be invoked from the editor, CLI, Workbench or Arexx. All internal tables are configurable for RAM management. REAL and LONGREAL support via the Amiga's library code allows use of hardware floating point boards. The compiler can generate debug information for a machine level debugger. It also generates inline calls to the Amiga's operating system, obviating the need for stub routine libraries. Termination procedures for each module are said to allow easy cleanup of resources.

The editor is optimized for Modula-2 source code production and features: an auto-case–correct that converts Modula-2 keywords to their correct case; a word completion capability that allows long names to be specified in a dictionary that is used to complete words when enough unique characters have

been typed; and display of compiler syntax errors in the source. The debugger shows Modula-2 source at point of program error and displays the contents of variables. Cost of M2Sprint is $385 US, plus shipping and handling. A demo disk is available. Write to: M2S, Inc., P.O. Box 550279, Dallas, TX 75355, USA. (214) 340-5256.

Dual serial board supports multiple users: ASDG's dual serial board occupies a single expansion slot in the A2000 and provides two AT-compatible 9-pin D connectors. It does not interfere with the original built-in serial port. The board comes with a disk containing a fully compatible driver and handler that permits simultaneous access to three or more serial devices.

According to a spokesman, each port has a hardware limit of 200 Kbits per second. However, this rate is CPU-dependent; the spokesman indicated that, in practice, 76.8 Kbits per second was a more realistic maximum. The dual serial board joins the already-shipping *Twin-X*–based *SBX-Serial/2* and *SBX-Serial/4* serial port devices. Cost is $299 US.

At the same time that ASDG announced its board, Oxxi, Inc. has stated that its *A-Talk III* terminal program is the first program to support the ASDG products. According to the company, A-Talk III and the ASDG dual serial board would make it possible to automatically research a stock portfolio from a Dow Jones BBS, book a plane reservation from OAG, and output a file to a serial printer, all simultaneously.

Version 1.1c of A-Talk III features a script language and ARexx support, and adds WXmodem and YmodemG protocols, an elapsed time clock, an improved phone book and support of both Telebit and U.S. Robotics high speed modems. Cost is $99 US. Registered owners of previous versions of A-Talk III can update directly from Oxxi, for a fee of $8 US.

For more information, contact: ASDG, Inc., 925 Stewart St., Madison, WI 53713, USA, (608) 273-6585; Oxxi, Inc., P.O. Box 90309, Long Beach, CA 90809-0309, USA, (213) 427-1227.

68020 accelerator board: The *Midget Racer* accelerator board from Computer System Associates (CSA) plugs into the 68000 socket of any Amiga and provides a 12-MHz 68020 running at 7.14 MHz. It also provides a socket that supports a 68881 or 68882 at speeds up to 33 MHz. Cost is $399 US, with a 68020 installed.

CSA claims speed increases of up to 60% over a stock Amiga; the math coprocessor provides further speed increases in such math-intensive applications as ray-tracing and computer-aided design.

For more information, contact: CSA, 7564 Trade St., San Diego, CA 92121, USA. (619) 566-3911.

Quarterback *V3.0* is a fake: Central Coast Software has announced that a purported version 3.0 of *Quarterback*, their backup utility program, is a fake. The imposter has apparently appeared on several Amiga pirate bulletin board systems. The current version of Quarterback is V2.2, which was released at the end of September, 1988.

The company is currently working on version 2.3 of Quarterback, which will backup to devices that don't have a standard mountlist entry. They are also working on Mac-2-DOS, a combination of hardware and software for reading and writing Macintosh files on the Amiga.

For more information, contact Betty Chamberlain of Central Coast Software, 424 Vista Avenue, Golden, CO 80401, USA. (303) 526-1030.

Lower price for C++: Claiming higher-than-expected sales, Lattice, Inc. has announced that, effective April 3, the prices of its *C++* package for the Amiga has been reduced from $500 US to $300 US. The company is also offering additional savings to current customers in the form of discount coupons. Owners of the Lattice AmigaDOS C compiler may take $100 US off the price of C++, and purchasers of C++ may save $100 US off the price of Lattice C. The coupons are included in Lattice's C++ and C Compiler packages. Registered users may also obtain the discount coupons through Lattice's customer service department.

Lattice has also added a new C manual to the C++ compiler that includes new documentation for the standard Lattice C library. The manual provides a synopsis of each C function's usage, a description of the function's purpose, the return value provided by the function and a cross reference to other related functions. The manual also gives numerous examples to help users see how these functions can fit into their application.

Lattice C++ is available from dealers and distributors worldwide or directly from Lattice. For further information or to order, contact: Lattice, Inc., 2500 south Highland Ave., Lombard, IL 60148, USA. (312) 916-1600.

Enhanced BASIC and source level debugger: New versions of Delphi Noetic Systems' *F-BASIC* compiler and source level debugger are now available.

Version 2.0 of F-BASIC adds: the ability to read or write a complete IFF file with a single command; random access file support; improved detection and processing of animation collisions; double-precision real numbers that includes support of math coprocessors; Workbench support for both the F-BASIC compiler and the programs it generates; and a linker that produces true stand-alone object files. Version 2.0 of the source debugger adds assembly breakpoints, 'multi-trace' and new display features.

For more information, contact: Delphi Noetic Systems, Inc., P.O. Box 7722, Rapid City, SD 57709, USA. (605) 348-0791.

More tools from Abacus: Abacus has released two books for Amiga programmers, and a disk of software tools.

Amiga 3-D Graphic Programming in BASIC is said to detail techniques and algorithms for writing three dimensional graphics programs. Users will learn the basics of ray tracing in all resolutions, light sources, shading and more. The book also features: an object editor for creating 3D objects in any resolution; IFF format graphics; many working examples; and an introduction to 3-D math for novices. Cost for the 300-page. book is $19.95 US.

A diskette is also available for the book, at a cost of $14.95 US. It contains AC/BASIC compiled versions of the EDITOR and *TRACER* programs, plus complete BASIC source code and a large collection of objects.

More Tips and Tricks for the Amiga, which is volume 12 in Abacus' Amiga Reference Library, contains techniques for using AmigaDOS and WorkBench 1.3 efficiently. Readers can learn how to make their programs more user-friendly using pulldown menus, sliders and tables. They can also learn how to: write a computer virus-checker; disable fast RAM; upgrade to a faster processor; use the NEWCON and PIPE devices; access machine language and C programs from *AmigaBASIC*; and so on. The 224-page. book costs $19.95.

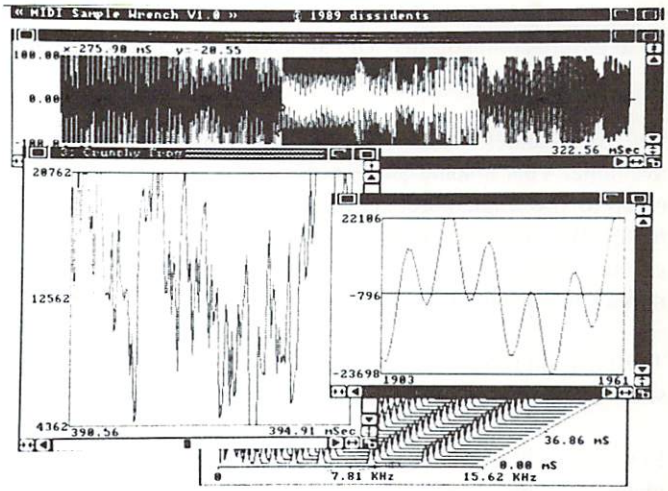
A partial list of the contents of the *Amiga DOS Toolbox* package includes a disk containing: *DeepCopy*, a disk copier that copies many different formats; *Speeder*, a disk speedup utility (not a disk cache); *Diskmon*, a disk editor; and a copy of the *AmigaDOS Quick Reference Guide*. Price is $59.95 US. Contact: Abacus, 5370 52nd Street SE, Grand Rapids, MI 49508, USA. Toll-free order line: (800) 451-4319.

Hypertext processor: *Thinker*, from Poor Person Software, is said to be one of the first products to bring hypertext to the Amiga. Hypertext combines word processing with database concepts; Thinker adds hypermedia and outline processing.

Suggested applications include: simple outline processing, word processing, an appointment calender, planning and writing of books, writing of documentation, organization of pictures, online documentation and program design (with links to the source via editor applications).

Version 1.03 of Thinker is now shipping. Version 2 will contain an ARexx interface, will use the AmigaDOS clipboard for cut and paste operations and may include some print formatting. This version is a major revision and is not expected before the end of the summer. The retail price of Thinker is $80 US. Contact: Poor Person Software, 3721 Starr King Circle, Palo Alto, CA 94306, USA. (415) 493-7234.

Sound sample editor: dissidents has announced the release of their *MIDI Sample Wrench* sound editing software. The Wrench works with 16-bit CD-quality sounds, which can be uploaded from MIDI-based samplers or disk files, extensively modified and then downloaded to other MIDI samplers. Full cut and paste editing is available, along with freehand drawing and sound mixing tools. Other functions include digital equalization, 50-point time-scalable envelope generation, signal compression, sampling rate conversion, arbitrary transfer function generation, crossfade looping, and others. Sounds may be pre-



viewed with the Amiga's internal voices, or downloaded to a sampler and remotely triggered from the Amiga keyboard. Sample size is limited only by available memory.

MIDI Sample Wrench will multitask with other programs and has user definable screen colors and resolution. Since sampler support is provided via libraries, new samplers are very easy to add to the system.

Present support includes MIDI standard dump devices such as the Sequential Prophet 2000/2002, Yamaha TX 16W and others. Other samplers will follow. System requirements: any Amiga with 512 Kb of RAM (1 Mb or more preferred), MIDI interface and one or more supported samplers. US retail prices is $279.

For more information, contact Jim Fiore at dissidents, 730 Dawes Avenue, Utica, NY 13502, USA. (315) 797-0343.

Encryption, Mandelbrot set software from MegageM: MegageM is an independent developer of Amiga scientific and mathematical software, operated by Dr. Daniel Wolf. Dr. Wolf is the co-author of *Compute!'s Amiga Machine Language Programming Guide*. In addition to a variety of demo and freely distributable disks of scientific and mathematical software, MegageM also distributes two commercial products written by Dr. Wolf: *HAMandel* and *LuCypher I* and *II*.

HAMandel produces Mandelbrot and Julia set images using the Amiga's HAM mode to generate up to 256 colours. Version 2.0 is said to be substantially faster than the previous version and adds such features as:

- the ability to magnify the Mandelbrot set up to 10 trillion times due to new double-precision mathematics. This capability lets users produce images of areas that were previously inaccessible.
- automatic sensing of 68020 processor. Special math routines give a five to eight times speedup with any

68020-based Amiga.

- extended depth count menu options that permit evaluating each point in the fractal 511, 1023 or 2047 times
- saving of HAMandel images with an even number of bytes to ensure compatibility with all IFF image display tools and HAM paint programs

The HAMandel *AutoMAG* cell animator module is now available. This a stand-alone program for creating sequences of up to 100 Mandelbrot set images with set magnification of 6.66% By generating only small magnifications from one image to the next, a series created by AutoMAG can assembled into a smooth animation.

The program provides HAM mode on the Amiga to produce the same number of colours as the HAMandel program.

LuCypher I is a character-based file ciphering package. It simulates an eight-element rotor system and encrypts or decrypts files using a 16-character key.

LuCypher II is a complete RSA public key cryptosystem. The RSA method is widely considered to represent the most advanced means of data security currently available. Employing Knuth's variation of the RSA method, LuCypher II uses public and private keys with up to 300 digits.

Two programs are supplied with LuCypher II: *RSAky* generates sets of keys (public/private key pairs_) and *RSAed* encrypts and decrypts files. The RSA public key method also provides a means of authentication of message origin.

The HAMandel program costs $39.95 US; with the AutoMAG module, it costs $59.95 US. The AutoMAG module alone is available to registered users of HAMandel for $20.00 US. The LuCypher I program is $49.95; the LuCypher II program in $199.95.

Note that, due to American international traffic in arms regulations, the LuCypher programs are for sale in the USA and Canada only.

MegageM also offers two freely distributable demo disks: *HAMandel Animation Disk*, which contains a 600-Kb ANIM file showing a zoom animation of the Mandelbrot set made with the AutoMAG module; and *WolfPak1*, which contains 12 HAM fractal images made with HAMandel and Turbo Silver, and a variety of fractal, cellular automaton and prime number analysis programs. Each disk costs $5.00 US, plus $1.25 US for shipping. California residents add 6% sales tax.

MegageM welcomes submissions for publication of special purpose scientific and mathematical applications by authors who may believe their programs will be hard to publish due to their limited market.

For more information, contact: MegageM, P.O. Box 1785, Santa Maria, CA 93456, USA. (805) 349-1104.

(continued from page 76)

As a general purpose high-level language, GFA is powerful and efficient. When it comes to user-interface programming and other Amiga-specific issues, however, there are some major problems. It would seem that the language's port from the Atari version was done hastily and with little consideration for the Amiga way of doing things. For example, the reserved variables MOUSEK, MOUSEX and MOUSEY are used to read the mouse buttons and position. It appears that this is accomplished by reading hardware, since you can detect mouse clicks in the windows of other applications in the system! This is definitely *not* the way to do things on the Amiga, and the authors have obviously bypassed Intuition and the normal event stream altogether. This means that your GFA-BASIC programs will not cooperate properly with other programs in the Amiga's multitasking environment, which is something that BASIC should ensure.

There is also a method of getting input events from Intuition the way you would do it in C, by using IDCMP (Intuition Direct Communications Message Port) messages and the like. This was obviously a quickly-hacked solution to input, and requires that you specify hard-coded flags in your programs for IDCMP and window flags, then check for these bits in values returned by the MENU function. The documentation for all of this is meaningless for one who hasn't studied the Rom Kernel Manuals and learned how to program in C or assembler already, and even a skilled programmer will have difficulty in using this incredibly clumsy system for a real application. A simple program that responds to mouse events and the window's close gadget, for example, is non-trivial and requires a lot of messy code sprinkled with meaningless numbers. It is much easier to write properly Intuitionized programs in AmigaBasic, Hisoft BASIC, or just about anything else.

The Amiga-specific features are not all bad, however. Opening windows and screens couldn't be more convenient, and the graphics commands give you advanced capabilities like setting a 'clipping region' (a rectangle to which graphics are confined), and the powerful 'draw' command, which gives you the 'turtle graphics' commands found in the LOGO language. Full access to the Amiga's shared libraries is provided, and the functions in all the libraries are listed in the manual, along with brief descriptions and required parameters. Most of these functions, of course, don't make sense within the BASIC environment, but at least the extra power is there for those that know what they're doing.

Animation is supported as in AmigaBasic, which is a reasonable system for handling BOBs (Blitter Objects). Sound generation also works as in AmigaBasic, but this area seems to still have a few problems: our tests produced some unexpected results, followed by a system crash.

Overall, as a general-purpose high-level language with an interactive and fast environment, GFA BASIC 3.0 is quite good. Unfortunately, its poor interaction with the Amiga's user interface and terrible documentation are serious flaws that will have to be improved before it can be recommended.

Reader Service Card

Fill out this card (please print) or attach your peel-off mailing label, and circle the reader service number for the ads that you'd like more information about (see Advertiser Index below).

Name _____

Company _____

Address _____

_____

Postal/Zip Code _____

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

TA V2I4

Advertiser	Page	Reader Service #
Antic Software	4	1
ASDG Inc.	28	2
Checkpoint Technologies	28	3
CSA	75	4
Electronics 2001	73	5
Fishers Computers & Software	73	6
Forethought	75	7
Gramma Software	70	8
King Publishing	28	9
M.A.S.T.	1	10
Metadigm Inc.	7	11
Michtron	Outside Back Cover	12
Microway	Inside Back Cover	13
Newtek Inc.	70	14
OTG Software	75	15
Pixelations	41	16
R&DL Productions	73	17
Rainbows Edge Productions	61	18
Ramco Computers & Software	75	19
Safe Harbor Computer Supplies	61	20
The Puzzle Factory	7	21
Vidtech International Inc.	Inside Front Cover	22
Virtual Reality Lab Inc.	41	23

place
stamp
here

Transactor
10-85 West Wilmot Street
Richmond Hill, Ontario, Canada
L4B 1K7

Advertising in Transactor for the Amiga

Interested in advertising in *TransAmi?* Call
Mike Grantham at 416/764-5274

for the AMIGA Transactor

The Magazine for Amiga Programmers

Please print your name and address below (or transfer your mailing label)

Check here for change of address ☐

Name: _____

Address: _____

City: _____ *Province/State:* _____

Zip/Postal: _____ *Phone:* _____ *CompuServe ID:* _____

NOTE: Prepayment required. Purchase orders will be accepted ONLY if accompanied by payment.

Subscribers only: please enter subscription number from mailing label ☐

Subscription Order Section

PLEASE NOTE THAT TRANSACTOR FOR THE AMIGA IS NOW PUBLISHED MONTHLY

	Canada ($Cdn.)			U.S.A. ($US)			Foreign ($US)			Air Mail ($US, overseas only)		
	Mag	Disk*	Both*	Mag	Disk	Both	Mag	Disk	Both	Mag	Disk	Both
12 issues	☐ 35.00	☐ 123.00	☐ 158.00	☐ 27.50	☐ 101.00	☐ 128.50	☐ 38.50	☐ 119.50	☐ 157.50	☐ 73.00	☐ 145.00	☐ 218.00
24 issues	☐ 65.00	☐ 220.00	☐ 270.00	☐ 49.00	☐ 160.00	☐ 190.00	☐ 69.00	☐ 220.00	☐ 290.00	☐ 135.00	☐ 250.00	☐ 390.00

*** Ontario residents please add 8% of disk subscription price for provincial sales tax. No PST on magazine subscriptions.**

☐ **Please send Microfiche instead of magazines** ☐ **Send me info on Transactor Classic (C64/128)**

Combo Subscription Bonus!

If you are subscribing to BOTH the magazine and the disk, check one of the following FREE gifts:

☐ **Transactor Classic Amiga Disk #1** ☐ **Fish Catalog Disk** ☐ **Extended Subscription (13 Issues)**

Product Order Section

Ontario residents please add 8% PST on all items

$Cdn.	$U.S.	
☐ 14.95	12.95	Transactor Classic Amiga Disk #1
☐ 14.95	12.95	Transactor Classic Amiga Disk #2

Fish Disks $6.95 US, $7.95 Cdn. each

Note: Disk numbers not yet released may be ordered in advance - simply add numbers to end of list. Disks 57, 80 and 88 are not available.

☐ 1	☐ 2	☐ 3	☐ 4	☐ 5	☐ 6	☐ 7	☐ 8	☐ 9	☐ 10	☐ 11	☐ 12	☐ 13	☐ 14	
☐ 15	☐ 16	☐ 17	☐ 18	☐ 19	☐ 20	☐ 21	☐ 22	☐ 23	☐ 24	☐ 25	☐ 26	☐ 27	☐ 28	
☐ 29	☐ 30	☐ 31	☐ 32	☐ 33	☐ 34	☐ 35	☐ 36	☐ 37	☐ 38	☐ 39	☐ 40	☐ 41	☐ 42	
☐ 43	☐ 44	☐ 45	☐ 46	☐ 47	☐ 48	☐ 49	☐ 50	☐ 51	☐ 52	☐ 53	☐ 54	☐ 55	☐ 56	
☐ 58	☐ 59	☐ 60	☐ 61	☐ 62	☐ 63	☐ 64	☐ 65	☐ 66	☐ 67	☐ 68	☐ 69	☐ 70	☐ 71	
☐ 72	☐ 73	☐ 74	☐ 75	☐ 76	☐ 77	☐ 78	☐ 79	☐ 81	☐ 82	☐ 83	☐ 84	☐ 85	☐ 86	
☐ 87	☐ 89	☐ 90	☐ 91	☐ 92	☐ 93	☐ 94	☐ 95	☐ 96	☐ 97	☐ 98	☐ 99	☐ 100	☐ 101	
☐ 102	☐ 103	☐ 104	☐ 105	☐ 106	☐ 107	☐ 108	☐ 109	☐ 110	☐ 111	☐ 112	☐ 113	☐ 114	☐ 115	
☐ 116	☐ 117	☐ 118	☐ 119	☐ 120	☐ 121	☐ 122	☐ 123	☐ 124	☐ 125	☐ 126	☐ 127	☐ 128	☐ 129	
☐ 130	☐ 131	☐ 133	☐ 134	☐ 135	☐ 136	☐ 137	☐ 138	☐ 139	☐ 140	☐ 141	☐ 142	☐ 143	☐ 144	
☐ 145	☐ 146	☐ 147	☐ 148	☐ 149	☐ 150	☐ 151	☐ 152	☐ 153	☐ 154	☐ 155	☐ 156	☐ 157	☐ 158	
☐ 159	☐ 160	☐ 161	☐ 162	☐ 163	☐ 164	☐ 165	☐ 166	☐ 167	☐ 168	☐ 169	☐ 170	☐ 171	☐ 172	

The Transactor for the *AMIGA* Disk $12.95 US, $14.95 Cdn. each

(these disks correspond to issues of **Transactor for the *AMIGA***)

☐ Vol. 1, Iss. 1 ☐ Vol. 1, Iss. 2 ☐ Vol. 1, Iss. 3 ☐ Vol. 1, Iss. 4 ☐ Vol. 1, Iss. 5 ☐ Vol. 1, Iss. 6

☐ Vol. 2, Iss. 1 ☐ Vol. 2, Iss. 2 ☐ Vol. 2, Iss. 3 ☐ Vol. 2, Iss. 4

Add flat $2.00 shipping & handling for any number of items ordered from above box.

Back Issues, Microfiche: $4.50 U.S., $4.50 Cdn. (MF=Microfiche, filled box=SOLD OUT)

☐ V1#1 ☐ MF ■ V1#2 ☐ MF ☐ V1#3 ☐ MF ☐ V1#4 ☐ MF ☐ V1#5 ☐ MF ☐ V1#6 ☐ MF

☐ V2#1 ☐ MF ☐ V2#2 ☐ MF ☐ V2#3 ☐ MF ☐ V2#4 ☐ MF

Ontario residents please add 8% provincial sales tax for any items ordered from this box.

☐ Visa ☐ MasterCard ☐ Cheque/MO enclosed Cheque #_____ Dated ___/___/___ Amount _____

Acct. # _____ Expires ___/___ Signature _____

Please allow 6-8 weeks for delivery in the U.S., 4-6 weeks in Canada.

FOLD ALONG SCORES WITH CORRECT REPLY CARD ON OUTSIDE. USE TAPE, PLEASE - NO STAPLES OR GLUE.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 390 BUFFALO, NY

POSTAGE WILL BE PAID BY ADDRESSEE

The Transactor
P.O. Box 338 Station 'C'
Buffalo, NY, 14209-9990

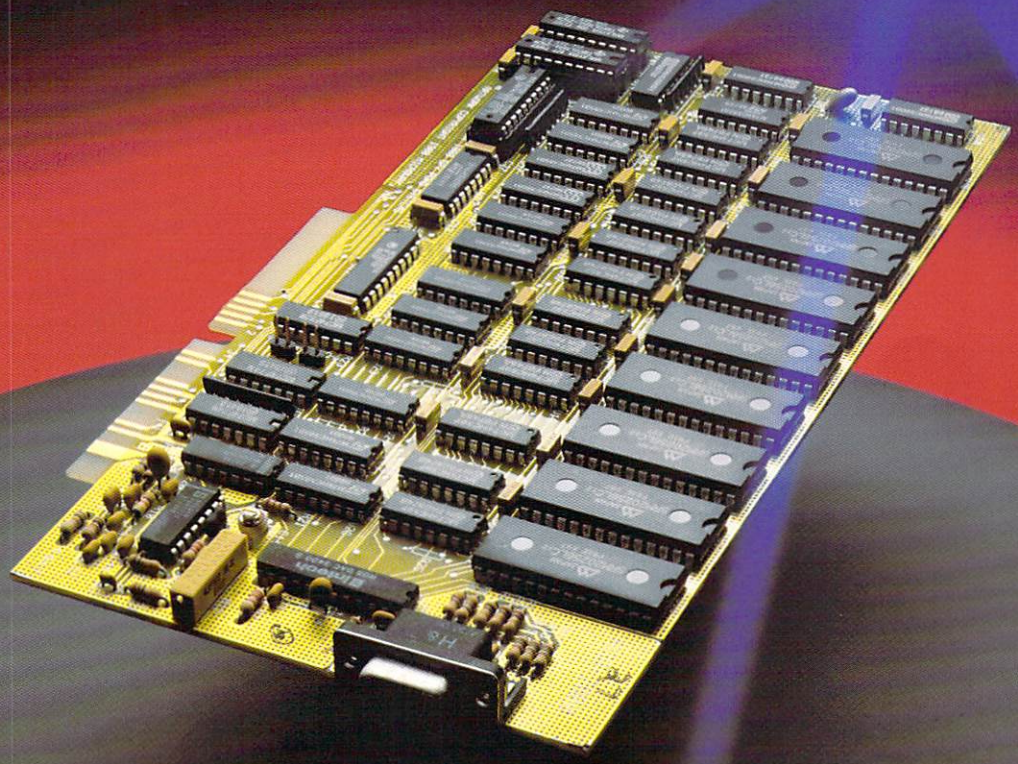
Business Reply Mail
No Postage Stamp Necessary
if mailed in Canada

Postage will be paid by:

Canada Post Postes Canada
46487

The Transactor
501 Alden Road
P.O. Box 3250
Markham Industrial Park
Markham, Ontario
L3R 9Z9

AGA 2000
The flickerFixer™



UNLOCK THE GRAPHICS POWER OF YOUR AMIGA 2000!

flickerFixer is an advanced graphics adaptor that eliminates your Amiga 2000's interlace flicker and visible scan lines. The result: superior quality color or monochrome graphics and text — for such demanding applications as CAD/CAM, Desktop Presentation, Graphic Design, Animation, 3D Modeling, Video, and Word Processing.

flickerFixer upgrades the Amiga 2000 with a flicker free 4096 color palette, has an overscan mode that features a screen size of 704 x 470 pixels and drives most of the popular PC Multiscan and VGA monitors, including the NEC Multisync and Mitsubishi XC1429C.

Accolades include: ***Best of 1988 Award,*** Commodore Magazine (12/88); ***#1 Reader's Choice Award***, AMIGAWORLD (12/88); ***"The display is fantastic . . . It is the best display we have ever seen on any computer system."*** Amiga GURU (5/88)

flickerFixer fits into the Amiga video slot, is fully compatible with all software, and does not modify the standard Amiga video signals. For more information or to order, call MicroWay Sales at (508) 746-7341 or your Amiga Dealer. Priced at $595, ***flickerFixer*** is made in the USA and is FCC Class B approved.

MicroWay . . . Respected throughout the industry for high quality engineering, service and technical support.

Micro Way

World Leader in PC Numerics

P.O. Box 79. Kingston, MA 02364 USA (508) 746-7341
32 High St., Kingston-Upon-Thames, UK, 01-541-5466
USA FAX 617-934-2414 Australia 02-439-8400

flickerFixer and MicroWay are trademarks of MicroWay, Inc. Amiga is a registered trademark of Commodore. Multisync is a registered trademark of NEC.

The Power to Create

HiSoft BASIC Professional is a fully-integrated compiler and editor that makes it easy to convert all your AmigaBASIC programs into fast-running machine code; in fact, it is so simple and quick to use that you will soon find that you no longer need the AmigaBASIC interpreter, you can do everything you want with **HiSoft BASIC Professional**.

Full support of the Amiga is included as standard, with extensive window, screen and graphics commands. Amiga libraries can also be accessed as if they were built-in statements allowing complete machine access.

HiSoft BASIC Professional includes full MENU support, with event trapping and powerful sprite routines, using the OBJECT keywords.

Compiled programs can execute in their own window(s) or use the CLI window for minimum size.

For larger programs and for those with strong editor preferences we also supply an editor-less compiler for traditional disk-to-disk compilation methods.

HiSoft BASIC Professional is a no-limits language; string variables may be up to 16Mbytes in length and there are no limits on array sizes either (subject to available memory). Code generated is fully 68010/020/030 compatible.

Compiled programs have no run-time overhead. Compiled programs multi-task, of course, but as they all share the same library, memory usage and disk space is kept to a minimum. There is also an option to produce completely stand-alone programs which include only as much of the library as they need.

Features include:

✓ Compatible with Microsoft QuickBASIC 3 on the PC, Atari ST BASIC, AmigaBASIC and most other BASICs,

✓ Full structured programming.

✓ Optional line numbers, lots of compiler and run-time options allowing tremendous flexibility both at compile & execution time,

✓ Full recursion in sub-programs and functions plus local and global variables, arrays as parameters,

***HiSoft BASIC Professional** - now BASIC programming on the Amiga has come of age.*

Order your Copy Today at $159.95, and take a look at Devpac Amiga, the best Assembler available for the Amiga. Together they make an unbeatable programming team!

	HiSoft Professional™	AC BASIC™
SIEVE		
Compile time (secs)	7	20
File size (bytes)	868	2228
Run time (secs)	1.0	2.7
CALC		
Run time	1.6	4.9
BSPREAD		
Compile time	55	80
File Size	35816	56036

DevPac Amiga:
The Assembler of choice

HiSoft Devpac Amiga Version 2 is the latest version of the acclaimed assembly language development tool for the Commodore Amiga. It combines a full 68000 macro assembler coupled with a full screen editor, **GenAm,** and a powerful disassembler and debugger, **MonAm**. Devpac is designed to give you maximum power and flexibility, while remaining easy to use.

GenAm includes a full screen editor, running under *Intuition* and using the window- and mouse-oriented environment. The editor features a file requester, dynamic block handling (including delete, copy, save and insert from disc) a comprehensive range of Search and Replace commands and edit-time tab handling. It has a powerful, fast 68000 macro assembler that can assemble source code at a maximum speed of 75,000 lines per minute (on a standard A500). A stand-alone version of the assembler is supplied for users with strong editor preferences or who use batch files.

MonAm is a comprehensive, highly advanced symbolic monitor, disassembler, and debugger. It is multi-tasking, and you can set breakpoints in your code while it executes and stop runaway programs.

To make program development fast and easy, both the assembler and the debugger are available instantly from within the editor.

Order today, and pay just $99.95 for the best Assembler/Debugger for the Amiga!

MichTron®
576 S. Telegraph, Pontiac, MI 48053
Orders and Information (313) 334-5700